ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

IMPRESSO-LOAD:

BUILDING A KNOWLEDGE GRAPH FROM A LARGE-SCALE CORPUS OF HISTORICAL NEWSPAPERS

EPFL COURSEWORK

Julien Salomon
Supervised by: Matteo Romanello, Maud Ehrmann, Andreas Spitz, Frederic Kaplan

13th January 2021

CHAPTER 1

INTRODUCTION

1.1 CONTEXT & MOTIVATION

Historical newspapers are a great source of information, helping experts and historians dig some information on the past. Impresso, an ongoing, collaborative research project conducted by the *EPFL-DHLAB*, *ICL-UZH* and *C2DH*[1], has been building a large-scale corpus of **digitized newspapers**: it currently contains **81 newspapers** from Switzerland and Luxembourg (written in French, German and Luxembourgish), constituting a collection of almost 40'000'000 articles, for dates ranging from 1798 to 2018. This corpus was enriched with several layers of semantic information such as topics, text reuse and **named entities** (persons, locations, dates and organizations). The latter are particularly useful for historians as co-occurrences of named entities often indicate (and help to identify) historical events.

To make the best use of such a large knowledge database, building interactive tools such as the one Impresso has been developing, requires an underlying **document model** facilitating the search for historical events and the relations of entities such as persons or locations surrounding such event from the documents in an efficient and intuitive manner. **LOAD** is a **graph-based document model**, developed by Andreas Spitz and Michael Gertz [2], that supports browsing, extracting and summarizing the corpus it modelizes, using **entities**: **L**ocations, **O**rganizations, **A**ctors and **D**ates.

The goal of this project is to continue the process of applying the LOAD model to the Impresso corpus, adapting the LOAD implementation for the databases of the Impresso corpus.

1.2 CONTRIBUTIONS

This project follows up the work of Isabelle Pumford [3]. Building upon her code base, originally based upon Andreas Spitz's code base, the following contributions were brought:

- For any set of newspapers from the Impresso corpus, write a program that can generate a LOAD graph.
- Find solutions and insights to scale up, in order to generate a graph from the entire Impresso corpus.
- Use the **EVELIN interface** [4], created by Spitz and Gertz, to visualize and browse the generated graphs.

[1] https://impresso-project.ch [1]

CHAPTER 2

IMPRESSO-LOAD

The LOAD model is an **entity-centric network** used to represent co-occurrences of named entities in the context of their surrounding **terms**. A graph is thus formed based on an implicit network model.

2.1 NODES IN THE IMPRESSO-LOAD MODEL

There are four types of nodes in the LOAD network: documents, sentences, named entities and terms.

2.1.1 NAMED ENTITIES

Named entities are linguistic units (often proper names) that have been identified by an annotator as entities of interest, based on the unit and its context in the text they are mentioned in. As mentioned in Chapter 1.1, in the LOAD model, there are 4 types of entities: **Locations, Organizations, Actors and Dates**. These are arbitrary and a subset of entity types that can be defined, and can thus be modified. In the context of the Impresso-LOAD model, only **actors** (persons) and **locations** have been used as entity types as the annotations for the other types were not available. To define the entities, named entity recognition (NER) was performed using EXPRESS [5] and entity linking and disambiguation was performed using the **AIDA-light** system [6]. An example of an entity is: **Jacques Chirac**: mentioned as "*President Jacques Chirac*", or "*Jacques Chirac*", with entity id "*aida-0001-50-Jacques_Chirac*".

2.1.2 TERMS

Terms are words contained in the corpus, that have not been linked to an entity. For example, "*tree*" is not a location, nor a person, thus it is a term. Every term in the corpus has been annotated with its part of speech (POS) tag indicating what type of word it is (noun, verb, preposition...). To avoid overpopulating the graph with too many term nodes, and to reduce the vocabulary, as the corpus contains billions of words, terms have been filtered to keep only a selection of POS tags in the Impresso-LOAD system: **nouns, numbers, proper nouns, verbs and adjectives**. These POS tags were selected as they give the most information on their context.

One thing to note: on the original LOAD network, done over the Wikipedia data, terms were stemmed before being stored, while for the Impresso-LOAD network, words were kept as they are mentioned in the articles. This is because the Impresso corpus contains newspapers written in Luxemburgish, and no stemmer exists for that language.

2.1.3 SENTENCES

Sentences are lists containing entities and terms, that from now on will be called **tokens**. As there are no efficient tools for sentence segmentation for the Impresso data, due to OCR (Optical character recognition) noise, **artificial sentences** were created, by splitting the documents into groups of 7 consecutive tokens.

2.1.4 DOCUMENTS

Documents are composed of sentences. In the original LOAD model, a document was a Wikipedia article. In the Impresso-LOAD model, a document is a **newspaper article**.

2.2 EDGES IN THE IMPRESSO-LOAD MODEL

The edges in this graph are weighted and undirected.

2.2.1 EDGE TYPES

There is an edge between two nodes x and y if:

- x is a sentence and y is a document and x is a sentence of the document y.
- x is a token (entity or term), y is a sentence and x is in the sentence y
- x and y are tokens and there exists at least one document for which x and y occur at most c sentences apart; where c is a positive integer. In the Impresso-LOAD model, $c = 5$.

The structure of the graph can be seen in Figure 2.1. As it can be observed, terms and entities form groups that are interconnected with each other based on the rules explained above. Sentences and pages act as a structure to the graph, as they are not connected with each other: there is no edge between two sentences, or between two pages.

2.2.2 EDGE WEIGHTS

The weight between any edge that have one extremity that is either a document or a sentence, is binary (1 if the edge exists, 0 else).

The weight of any edge between **two tokens** is defined in the following equation:

$$w(x, y) = \sum_{i \in T_x, j \in T_y} \exp -\delta(i, j) \tag{2.1}$$

where T_x is the set of all the occurrences of the token x in the corpus and where $\delta(i, j)$ is the distance between the occurrence i of token x and the occurrence j of token y. Thus, $\delta(i, j)$ is equal to 0 if i and j are in the same sentence, then $\delta(i, j)$ becomes progressively larger until it reaches $c = 5$. Any value above 5 is set to $+\infty$.

2.3 BACKEND

Data storage is an important aspect of this project. In the original LOAD model, created on 4.6 million English Wikipedia articles, the data was stored into two different Mongo databases: one to contain the sentences of the corpus, and the other the annotated named entities.

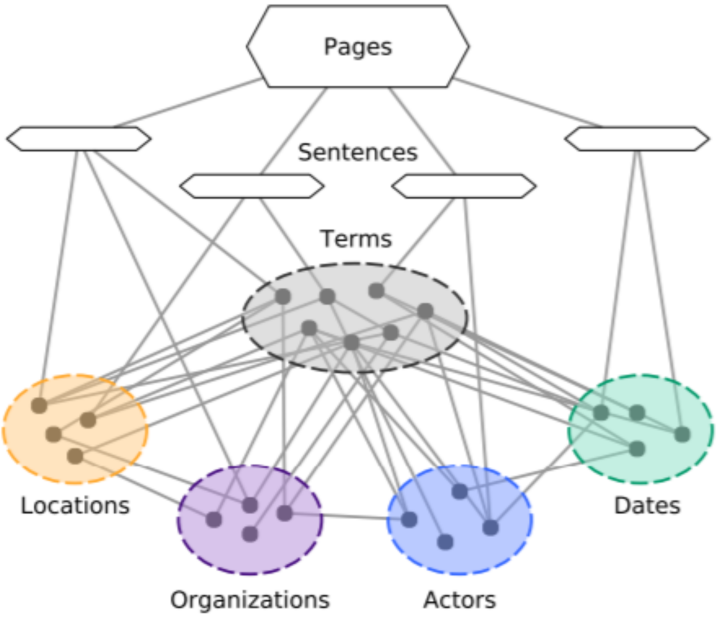


FIGURE 2.1
Schema of the LOAD graph structure (from [7]

In the case of the Impresso data, the data used to build the Impresso-LOAD network is separated into 3 databases:

- One S3 bucket, containing, for each combination of newspapers and year, the list of articles of all the issues of the newspaper for this year, and for each article, the list of **all the annotated words** contained in the article. The information, for each word is the part-of-speech tag, the offset (place in the article) and their lemmatisation.
- One *Solr* index containing **information about each article in the corpus**: its language, its title, its content (full scanned text of the article), its mentioned entities and their offset. Other information is contained but is irrelevant for this project.
- Another *Solr* database containing, for each mentioned entities in the corpus, **its entity information**: its label, its entity id, its wikipedia id (that links the entity to its wikipedia page) and other information irrelevant for this project.

One thing to note is that the final *Solr* database, mapping mentioned entities to their entity information has been created recently and that before it existed, the entities and their information were pulled from another S3 bucket that, similarly to the S3 bucket containing the annotated words of the corpus, contained, for each newspaper and year combination, the list of mentioned entities for each article, linked to their entity information.

2.4 LOAD GRAPH FORMAT

The LOAD graph, is created and represented in 15 *.txt* files: 7 files representing the different node types, (Locations, Organizations, Actors, Dates, Terms, Sentences and Pages), 7 files representing the edges, and one file summarizing the graph, giving metadata about the generated graph. The edges are written on

several files, one per node type: an edge is written to a file of one node type, if its *from* node is of this type (for example an edge from an Actor to a Term is written on the file *eAct.txt*).

2.4.1 METADATA FILE

The *metadata.txt* output file gives the following information:

- Number of pages with at least one entity
- Number of sentences with/without entities
- Number of entities in the corpus (total and per types)
- Number of edges (before and after aggregating the parallel edges)
- Number of nodes in the graph per type

2.4.2 VERTICES FILES

The files representing the vertices all have the same format. For each line the following data is given, all separated by a predefined separating character:

1. The label of the vertex. For terms, it is the term itself; for entities, it is their entity id; for sentences, it is their article ids followed by the index of their beginning and of their ending in the document; for pages it is the article id.
2. Degree of the vertex (how many edges are connected to this vertex), given the node type of the other node connected to it. The degrees are written in the following order: dates, locations, actors, organizations, terms, pages, sentences.

2.4.3 EDGES FILES

The files representing the edges all have the same format. They are being written by blocks of 8 lines each, each block representing the edges connected to a node of the type represented by the file. The first line is the node id. The seven following lines each represent a vertex type. For each of these lines, the vertex type is followed by a list of pairs of target node ids and edge weights.

CHAPTER 3

IMPLEMENTING THE IMPRESSO-LOAD GRAPH

3.1 GRAPH CREATION

Using the data described in chapter 2.3, it is now possible to generate the Impresso-LOAD graph.

The code base used to generate the graph is based on the code created by Andreas Spitz to generate the graph from the Wikipedia articles, and adapted it to the specificities of the Impresso corpus.

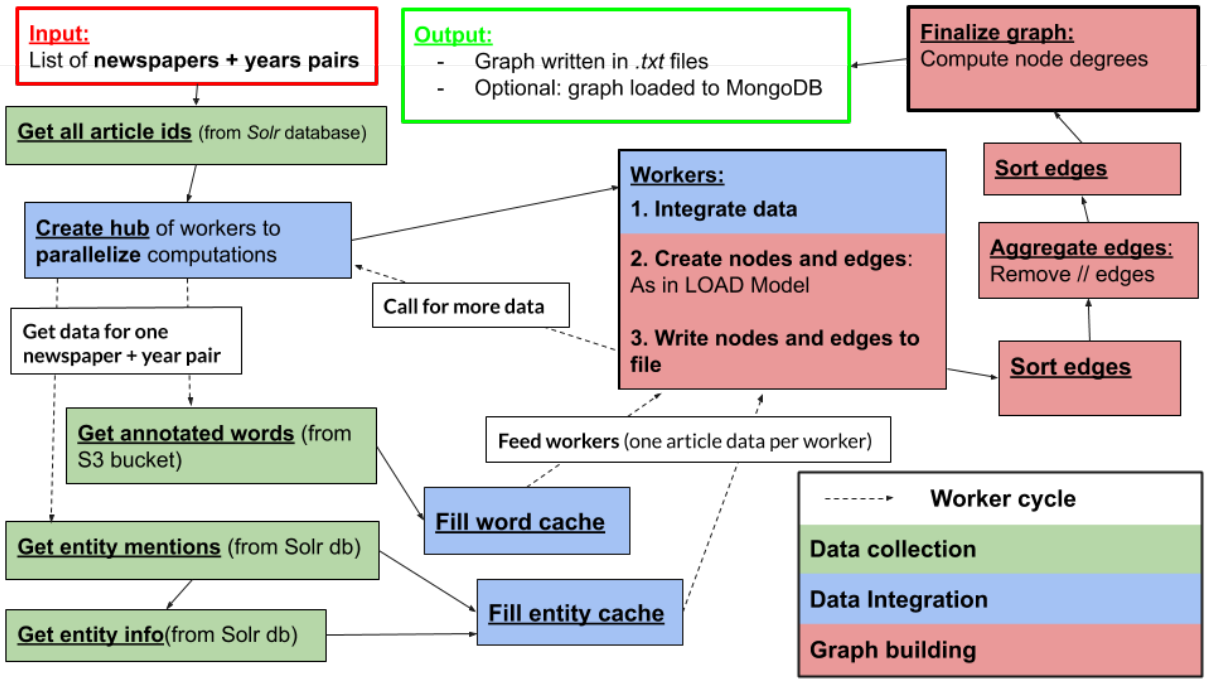


FIGURE 3.1
Graph creator code pipeline

The pipeline of the graph creation system is described in Figure 3.1.

3.1.1 INPUT

The input of the process is a combination of newspapers and years. If no year is specified for a newspaper, it is interpreted that all years must be queried.

For now, the code is being tested for a subset of the corpus. As the goal in the long term is to generate a graph for the entire Impresso corpus, no input will be required to run the process on the entire corpus (for all newspaper-year pair).

3.1.2 WORKER CYCLE

Once the input entered, the code has been designed to efficiently iterate the process of collecting the data, integrating it and building the graph.

Multiple **workers** work in parallel, and are synchronized by a **hub**. This hub manages the data by distributing it to the workers sequentially. The data is pulled from the databases for one newspaper-year pair from the input, and stored in **caches** that work like hash maps: given a key (an article id), the data is returned. A worker asks for the data (all annotated words and all entity mentions and their entity information) for one article that has not already been read previously, integrates it in the code, and writes the nodes and edges related to this article. If a worker asks for more data, and all articles for a newspaper-year pair have been read, the hub queries the data for another newspaper-year pair, while the workers either finish handling the article they are currently working on, or await the data.

The use of parallel workers is justified, so that pulling the data and storing it to the caches, which takes some time, is not a bottleneck for the code, as even if some workers are awaiting for the data, most would be working on articles in parallel.

3.1.3 PREPARING THE OUTPUT

As mentioned in part 2.1, nodes appear in the graph once per term and once per entity, even if the tokens are mentioned multiple times in the corpus. During the workers cycle (see section 3.2.2), when edges are being written, **parallel edges** are written. Edges are defined as parallel edges if they share the same connection between two nodes, even if their weights are different.

As explained in section 2.2.2, in the final LOAD model, parallel edges do not exist. The parallel edges are thus merged by summing their weights. To accelerate the process of finding parallel edges, the edges are first **sorted** before being **aggregated**.

Once the edges have been aggregated, the node degrees are computed, and the graph is finalized and written in an output folder. It is possible to also write the graph to a Mongo database, to render the graph compatible with the EVELIN interface (see chapter 5) under the format described in chapter 2.4.

3.2 DOCUMENTATION

As this is an ongoing project, and as it has a multitude of parts, newcomers are likely to work on the same code base. An emphasis on documentation has been put on the implementation of the graph, starting with the pipeline described in Figure 3.1.

The *readme* file of the github for this project[1] contains information that aims to simplify the work of future contributors, as well as reduce the adaptation time required to get familiar with this code base. The following information can be found:

[1] `https://github.com/dhlab-epfl-students/impresso-LOAD-salomon`

- **Code structure**: contains information on each package of the project, as well as detailed information on the most important Java classes in the code.
- **Information on how to run the code**: required libraries, input structure, useful command line tools. There is information on how to run the code on the cluster as well.
- **Information on how to launch the EVELIN interface** is also given (see chapter 5 to know more about the interface).

As well, some prompts can be activated modifying some boolean values in the code. These prompts give information about the run of the code, each with a different degree of information (information about which step of the code is currently running, information about the data being treated, information about the runtime of some processes of the code).

3.3 IMPROVING THE PIPELINE

As seen in Figure 3.1, the first thing done in the pipeline is to pull all of the article ids for each newspaper-year pair and store them in text files. This part of the pipeline is not necessary, as in the code, all of the articles are pulled for each newspaper-year pair; there is never a selection. If it is required to get some statistics on a newspaper-year pair (number of articles for example), this can be done by querying the specific information on the Solr index, rather than pulling all of the ids and then computing the statistics. Thus this aspect of the pipeline is skippable and should be deleted from the pipeline.

CHAPTER 4

SCALING UP

One of the main objective of the Impresso-LOAD project is to obtain a LOAD graph on **the entire Impresso corpus**, containing around 40 million articles. Runtime and memory management are thus important aspects of the code.

4.1 MEMORY MANAGEMENT

There are no major issues related to memory in the program.

When filling the caches with data (see chapter 3.1.2), a large *.jsonl* file is queried to get all the annotated words for a newspaper-year pair. As well, all entity mentions and their entity information are being queried and stored to another cache. These caches have a predefined maximum size, and handle their memory in a FIFO (First In First Out) manner, once the cache is full.

This data querying is the part that necessitates the most memory in the code. For the graph creation, a buffered writer writes to the different output files (see chapter 2.4), thus, the graph information is being stored in files while the code is running. The total output size (uncompressed) of the original LOAD graph done on the entire English Wikipedia corpus was of 70 GB [7], meaning that some memory needs to be made available to store the final output.

4.2 RUNTIME MANAGEMENT

Runtime is a much larger issue than memory for this project. With the latest main code update, concerning the fetching of the entity data using Solr (see chapter 2.3), generating the graph for the *Gazette de Lausanne* issues from 1986 to 1996 (included) takes approximately 17 days.

For information this represents 384'553 articles out of the 24'505'273 total articles of the entire corpus (taking out advertisements, obituary pages, tables, weather news and other unclassified content). Thus it takes 17 days to obtain 1.5% of the entire Impresso-LOAD graph.

Some runtime analysis has been conducted to obtain information on what aspects of the code are slowing it down. The values obtain are relative to the computer that ran those operations. For information, unless mentionned otherwise, the tests were done on a *MacBook Pro*, with a 6-Core Intel Core i7 and a 16GB memory.

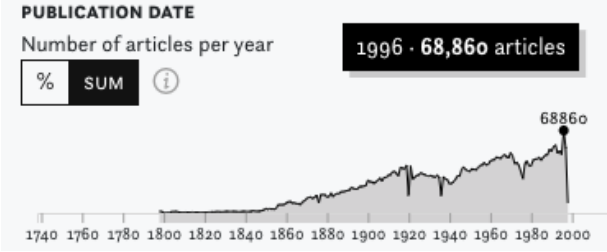


FIGURE 4.1
Number of articles from *La Gazette de Lausanne* (GDL) per year in the Impresso corpus

4.2.1 FETCHING DATA

ANNOTATED WORDS

The annotated words are queried by newspaper-year pair, in a zipped *.jsonl* file stored in an S3 bucket. Each line contains a *JSON* document containing, for each article in the newspaper-year pair, a list of sentences each containing a list of annotated words. Each JSON document is stored in the word cache (see Figure 3.1) with the article id as a key.

The largest newspaper-year pair tested is for the articles from the *Gazette de Lausanne* in 1996 (see Figure 4.1). Getting the file from the S3 bucket, reading it, and storing it to cache takes approximately **10 minutes** (depending on the computer running the program) for 36085 sentences, 723852 words.

The query time of the file itself is hard to measure as it relies on the Internet connection available. Nevertheless, for the tests realized, on the largest file queried (*Gazette de Lausanne* - 1996), the query took less than a second (trivial compared to the complete runtime of the operation).

The storage of the *JSON* documents in the cache is what takes up most of the runtime. A linear relationship between the number of words in a sentence, and the time taken to store the JSON file representing the words in the sentence can be observed in Figure 4.2. This graph was done querying 3 files: GDL-1996, GDL-1890, GDL-1798.

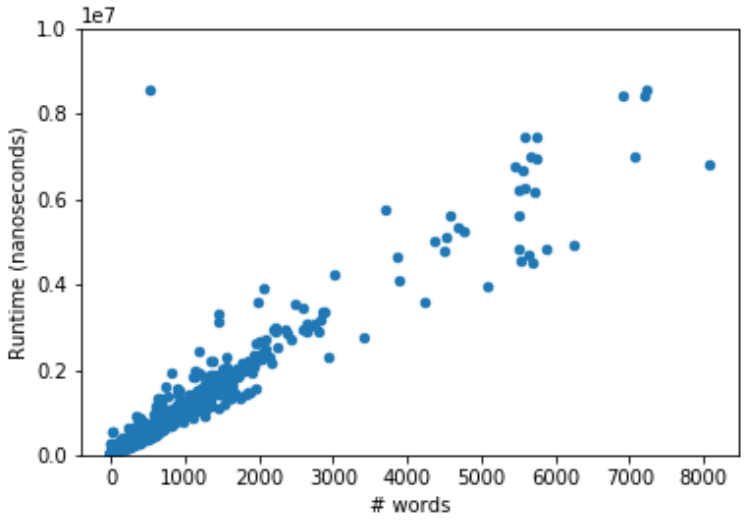


FIGURE 4.2
Runtime in nanoseconds of the storage of a sentence JSON in the cache given the number of words in the sentence

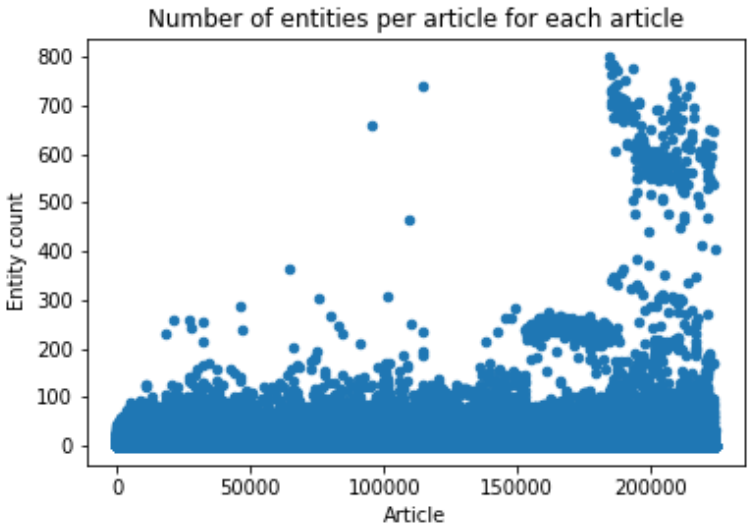


FIGURE 4.3
Number of entities per paper ordered from oldest to most recent for articles for every 10 years from 1806 to 1996

MENTIONED ENTITIES

Querying each mentioned entities for a given newspaper-year pair, happens in two steps.

First, all mentioned entities and their offsets grouped by articles for the newspaper-year pair are queried by bundles of 5000 from a Solr index. For each mention, the entity information is queried from another Solr index, indexed by the entity mention, the entity offset and the entity article id. The entity information queried is the **entity id**, the **entity label** and the **system** that identified the entities.

The processing of a document for getting all the entity information from *GDL-1798* and *GDL-1799* is of 0.1 seconds per document on average. Multiplied by the number of articles in the corpus, this amounts to a total query time for entities of a bit less than 45 days. This is without the fact that the number of entities per article gets larger for more recent years (see Figure 4.3).

4.2.2 NUMBER OF THREADS

The worker cycle is the central process of the graph creation. The number of workers working in parallel thus highly influences the runtime of the code. A grid search has been done by running the code with different numbers of workers (threads), by comparing the runtime of the code minus the data query time (as it is unrelated to the number of threads), to get the optimal value for threads.

The result values for the runtimes given the number of threads can be observed in Figure 4.4.

This result depends on the number of cores of the processor. This grid search should be redone for any new machine wanting to optimize this code. On the machine the test was performed on the optimized value is of 50 threads.

4.2.3 WORKER CYCLE

The worker cycle takes up most of the runtime of the code. A runtime analysis has been done on the worker cycle for all articles from *Gazette de Lausanne* in 1798 and 1799, representing a total of **2591 articles**.

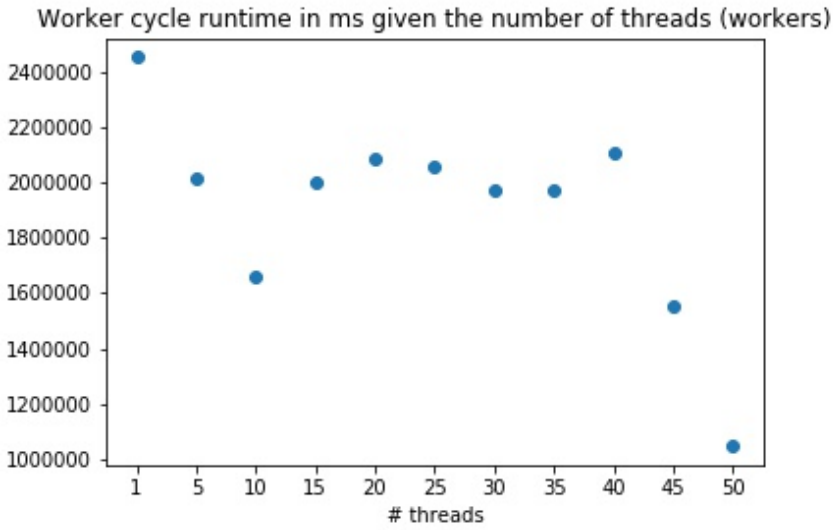


FIGURE 4.4
Runtime in milliseconds given the number of threads

First of all, the mean runtime for the processing of one article is of 15.2 seconds (median value of 11.1 seconds), with a large variance (see Figure 4.5). The total runtime, subtracting the time taken to fetch the data, is of **13.2 minutes** thus a render of **0.3 seconds per article**.

Multiplying this value to the number of articles in the corpus ($24,505,273 * 0.3$), we obtain a total runtime of a little more of **85 days** only counting the worker cycle (the time taken to obtain the data is not taken into account). This is supposing that the runtime is not related to the number of entities in NOT DONE

Again, it is important to note, that this runtime is specific to the computer the tests were performed on, and on another device, could be completely different. As well, as Figure 4.3 shows, the number of entities per article per year gets larger for more recent years, adding to the runtime even more.

Thus this part of the code is a bottleneck. To find out where the runtime comes from, the worker cycle process, for one article has been divided into multiple steps:

1. **Content item creation**: create an instance of a "content item" that will contain all relevant data. When created, the instance is empty, except for the **article id**.
2. **Inject linguistic annotations and entities**: gets data of the article from the word cache and the entity cache (see Figure 3.1) and injects it into the Content Item created.
3. **Sort the tokens:** the sorting function sorts the tokens (terms and entities) in function of their offset, putting them in the order they are placed in the text of the article.
4. **Get the tokens:** get the tokens as a list from the Content Item
5. **Create sentences:** create the artificial sentences (see Chapter 2.1.3), and add edges between the created sentences and their articles, between the tokens and the sentence they are in and edges between all tokens of the same sentence.
6. **Sort sentences**: so they are ordered as they appear in the text.
7. **Add edges between linked entities:** for every token pair (no duplicate and if (a, b) is a pair of tokens then (b, a) is a duplicate of that pair), check their distance (in terms of number of sentences between them), and add an edge between the tokens if the number of sentences between the two

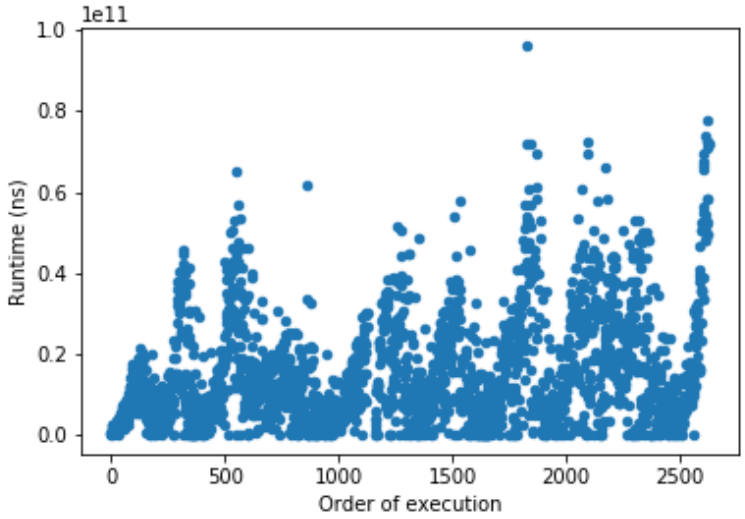


FIGURE 4.5
Runtime in nanoseconds of the processing of all articles from GDL 1798, 1799, when there are 50 threads running in parallel

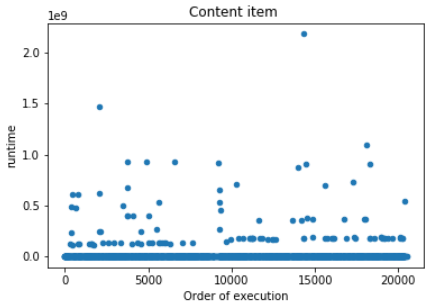


FIGURE 4.6
Runtime (ns) of the **Content item creation** step for articles in GDL 1798, 1799

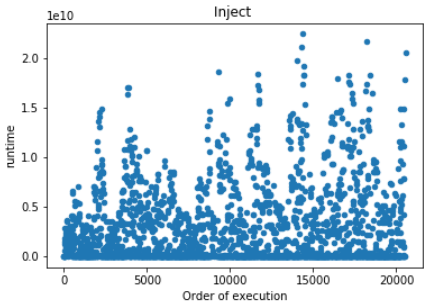


FIGURE 4.7
Runtime (ns) of the **Inject** step for articles in GDL 1798, 1799

tokens is lower than the maximum distance. In our case, the distance if of 5 sentences (see Chapter 2.2.1).

8. **Write edges:** once every edge is added, send them to the buffered writer in the hub that will write them to file.

The runtime for each of these steps has been plotted in the figures 4.6 to 4.13. There are no obvious bottlenecks as all steps have the same order of magnitude, except for some outliers in Figure 4.13. The outliers might be due to some computer related issue, as they appear consecutively in the order they have been handled.

When looking at the Figures 4.5 to 4.13, no true bottleneck has been identified in terms of articles, or in terms of parts of the process.

One way to accelerate the runtime would be to create the sentence in order, so that step 6 could be avoided, and step 5 and 7 could be merged, by keeping in memory, the n past sentences that have been created, n being the maximal distance between sentences, and writing the edges sequentially.

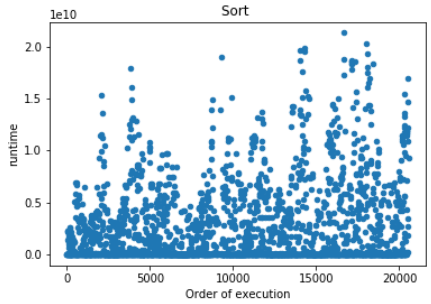


FIGURE 4.8
Runtime (ns) of the **token sorting** step for articles in GDL 1798, 1799

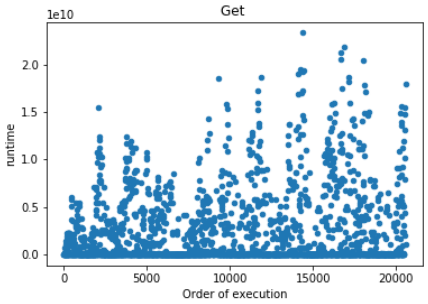


FIGURE 4.9
Runtime (ns) of the **get tokens** step for articles in GDL 1798, 1799

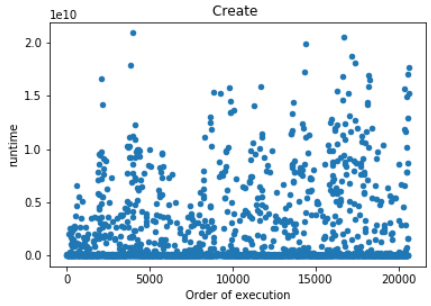


FIGURE 4.10
Runtime (ns) of the **sentence creation** step for articles in GDL 1798, 1799

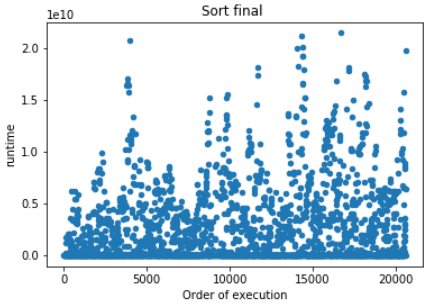


FIGURE 4.11
Runtime (ns) of the **sentence sorting** step for articles in GDL 1798, 1799

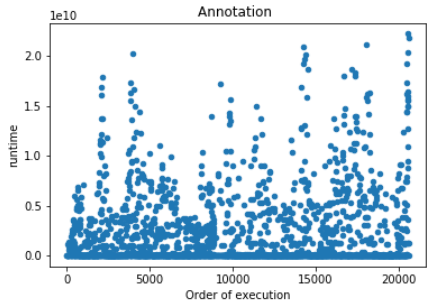


FIGURE 4.12
Runtime (ns) of the **edge creation** step for articles in GDL 1798, 1799

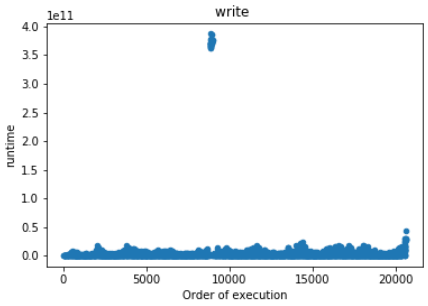


FIGURE 4.13
Runtime (ns) of the **edge writing** step for articles in GDL 1798, 1799

Another way would be to reduce the context size (maximal distance between sentences).

4.3 RUNNING THE CODE ON THE CLUSTER

Running the code on a cluster is a way of accelerating the runtime. The code has a *boolean* that should be set to *true* to change some properties of the code to accommodate the cluster settings, mainly the number of threads and the output folders.

By comparing runtimes, the cluster does not highly accelerate the process of running the code on the cluster. It does provide stability compared to a personal computer, and is the method of choice to run the code on a large portion of the corpus.

CHAPTER 5

INTERPRETING THE GRAPH

When obtaining the graph output (described in chapter 2.4), it is difficult to debug, browse or interpret.

5.1 GRAPH VISUALIZATION

To simply visualize the graph to see if it does have the desired format (see Figure 2.1), a *Python* script has been written and can be found in the github[1] repository to generate an image of the graph.

What can be seen in the Figures 5.1 A and B, is that the graph becomes rapidly unreadable due to the many nodes and edges. It is a decent enough tool to verify that the structure of the graph is logical, as some information can even be observed by hand, if a graph is generated for a small article (5.1 A) it is more or less readable, bur rapidly becomes problematic to read for a longer one (5.1 B).

As well it does not help to browse or to interpret in any ways the corpus. A more interactive a browsable tool should thus be used.

Generating this graph also shows the importance of being able to **browse the graph** as visualizing it in its entirety would not allow anyone to get any information out of it.

5.2 EVELIN INTERFACE

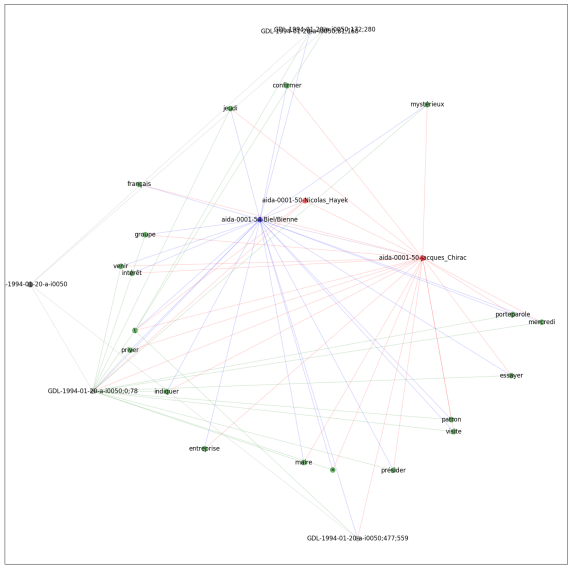
The EVELIN interface, created by Spitz and Gertz [4], is an interface allowing its users to browse through a LOAD graph.

The interface works in the following manner: given a list of input entities, generate either a list of related entities, terms, sentences, or documents related to this list of entities, and **sorted by score**, defined by the weights of the edges of the graph. The interface can also generate, for such a query, a subgraph, linking the inputed list of entities to their best nodes defined by the weights of the edges of the graph as well.

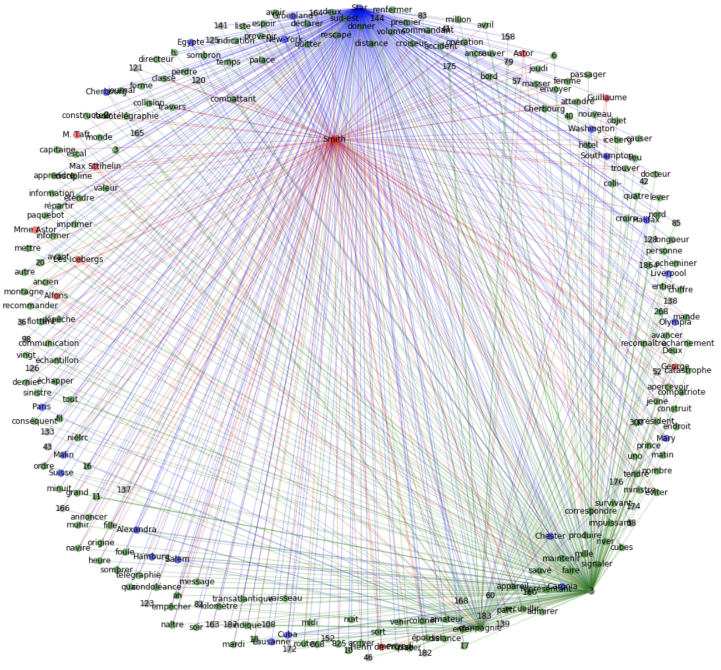
Figures 5.2 to 5.8 are screenshots from the EVELIN interface implemented for the LOAD graph generated based on all articles from *La Gazette de Lausanne* from 1986 to 1996, and for the search input *Jacques Chirac*, the entity representing the French politician Jacques Chirac.

When selecting input queries, the search bar suggests possible answers. This functions as the EVELIN backend has a lookup function that returns all entities that contain the **word** or the **words** written in the search bar. It does not work with sub-texts, only with full words. For example, "Jacques Chirac" only

[1] `https://github.com/dhlab-epfl-students/impresso-LOAD-salomon`



(A) LOAD graph generated for article "*SWATCHMOBILE, Jacques Chirac viendra l'essayer" (87 words)*



(B) LOAD graph generated for article "*Le naufrage du Titanic" (1200 words)*

FIGURE 5.1
LOAD graphs for one article

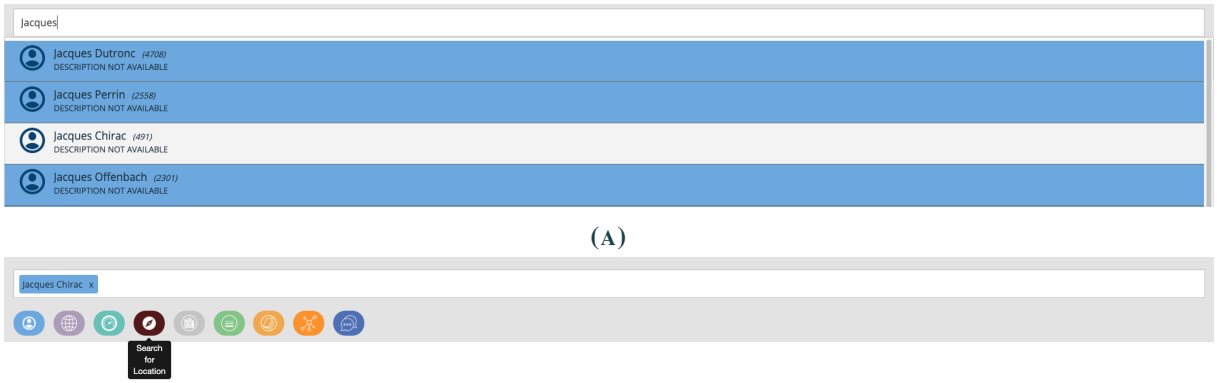


FIGURE 5.2
EVELIN search bar when typing "Jacques" (A) and when selecting "Jacques Chirac" (B)

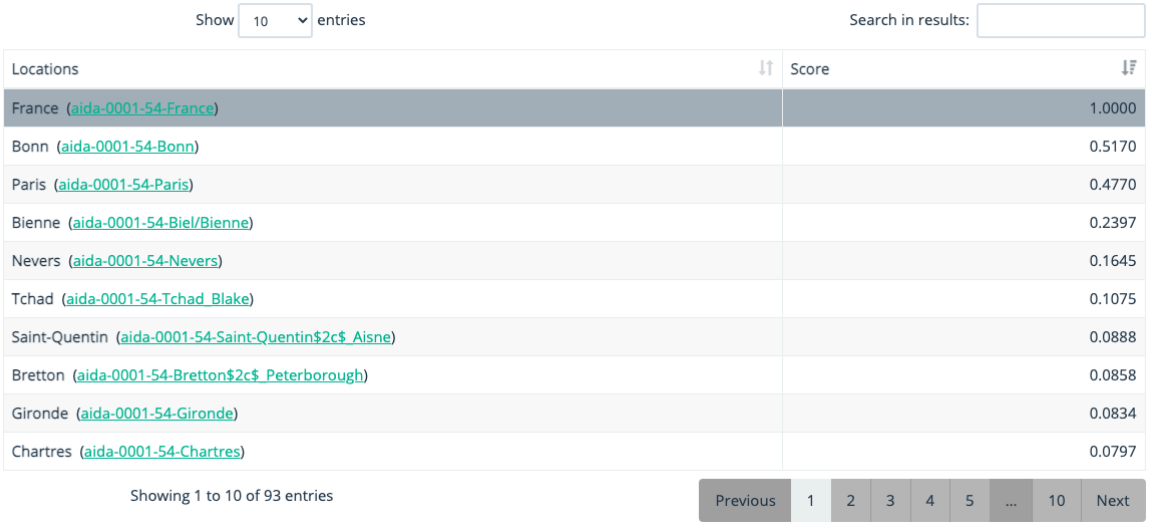


FIGURE 5.3
EVELIN page result when querying locations related to actor entity Jacques Chirac

appears in the suggestions (see Figure 5.2a) if the word "Jacques", "Chirac" or "Jacques Chirac" are written. If "Chira" is written, the output "Jacques Chirac" does not appear in the suggestions.

Each line from the list of results for related entities (Figure 5.4 and 5.3), have the following format: "ENTITY LABEL (ENTITY_ID) SCORE". On the original EVELIN interface on the wikipedia data, the underlined entity id was a link linking the entity to its wikipedia page. Here the entity id is put as a placeholder to either be replaced with a link towards the Wikipedia page of the entity, or from another source depending where the project goes in the future.

Each line from the list of results for related terms (Figure 5.6) has the following format: "TERM SCORE".

Each line from list of results for related sentences (Figure 5.5) has the following format: "FULL SENTENCE (source) SCORE". The underlined "source" used, in the original EVELIN interface, to link to the Wikipedia page where this sentence was contained. Here, the link is undefined, but has as argument the article id of the article containing the sentence. It could, in the future, either link to the Impresso app page related to the article, or even to another page from the EVELIN interface printing the full article with additional information.

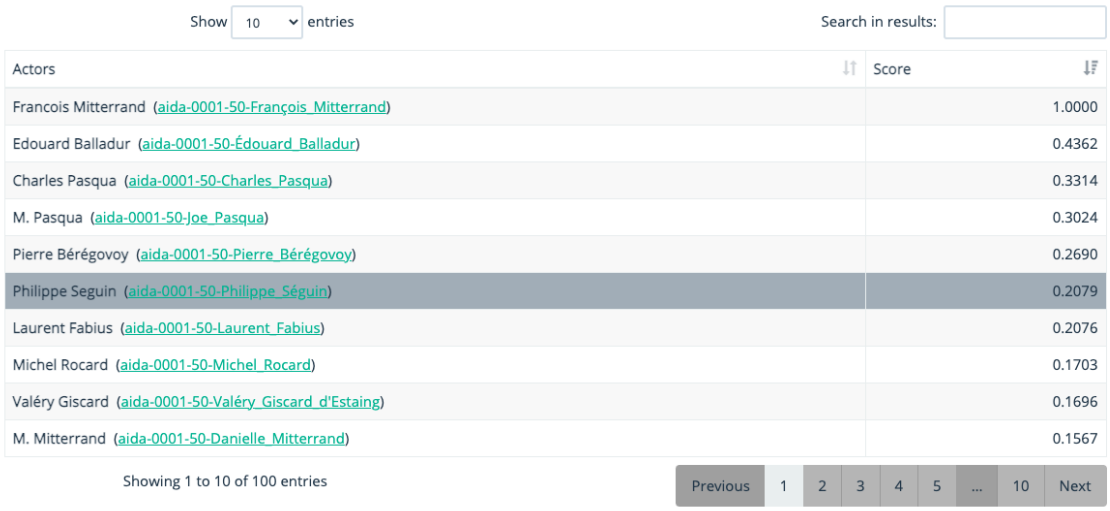


FIGURE 5.4
EVELIN page result when querying actors related to actor entity Jacques Chirac

Each line from the list of results for related pages (articles) (Figure 5.7), has the following format: "ARTICLE TITLE SCORE". Same as for sentences, here the link is undefined but has as argument the article id of the mentioned article and could link to another page in the future.

The subgraph result (Figure 5.8), shows the token (entity or term) nodes with highest score related to the search inputs.

The score is computed by ranking all targeted entities (all actors, all locations or all tokens) in relation to the entities in the input by summing the weights of the edges between all the input entities and the targetted entities, and by normalizing the scores for them to be between 0 and 1. Once the ranking of the tokens is done, one can group them by sentence and rank the sentences and then the pages, to determine the score of the results when searching for sentences and pages.

5.3 USAGE EXAMPLES

5.3.1 INTERACTIVE TOOL

Currently, it is possible to search and navigate through the Impresso-LOAD corpus using the interface by searching keywords in articles, and filtering using dates, topics, location or persons. The EVELIN interface does similar tasks, but using the underlying graph.

This brings **entity disambiguation** when searching for entities as well as **context introduction** in the search. Entities are only linked if they appear in the same context, defined by the edge weights (see Chapter 2.2.2) thus not necessarily if they appear in the same article.

As well it switches the focus of the searches from articles to any nodes in the LOAD graph (including articles), allowing the search on different aspects of the corpus.

One can imagine, once the entity definition is done for every entity type (Organizations, dates and even more) is done, that the EVELIN interface could be an efficient manner to *summarize* an entity or a combination of entities: "Jacques Chirac", would summarize the president Jacques Chirac's career as described in the Swiss newspapers, and Jacques Chirac, Switzerland, 1998 would certainly show the places he visited, and the people he met during Jacques Chirac's official trip to Switzerland.

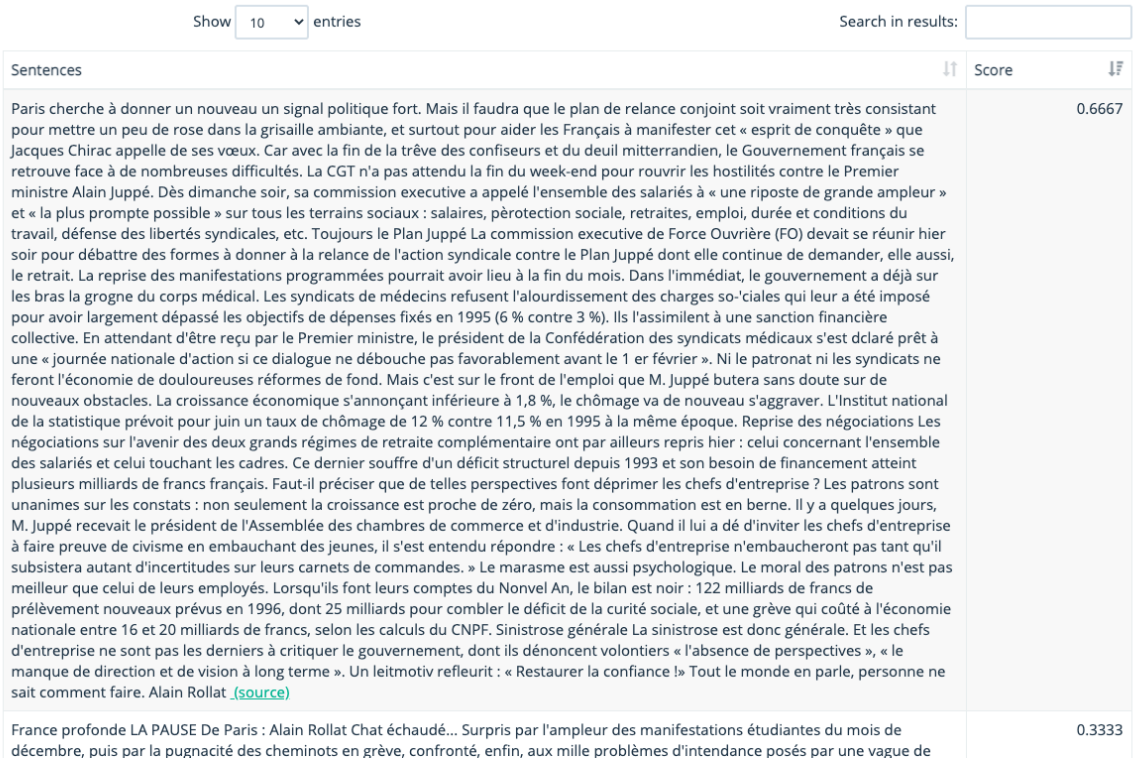


FIGURE 5.5
EVELIN page result when querying sentences related to actor entity Jacques Chirac

5.3.2 DEBUGGING TOOL

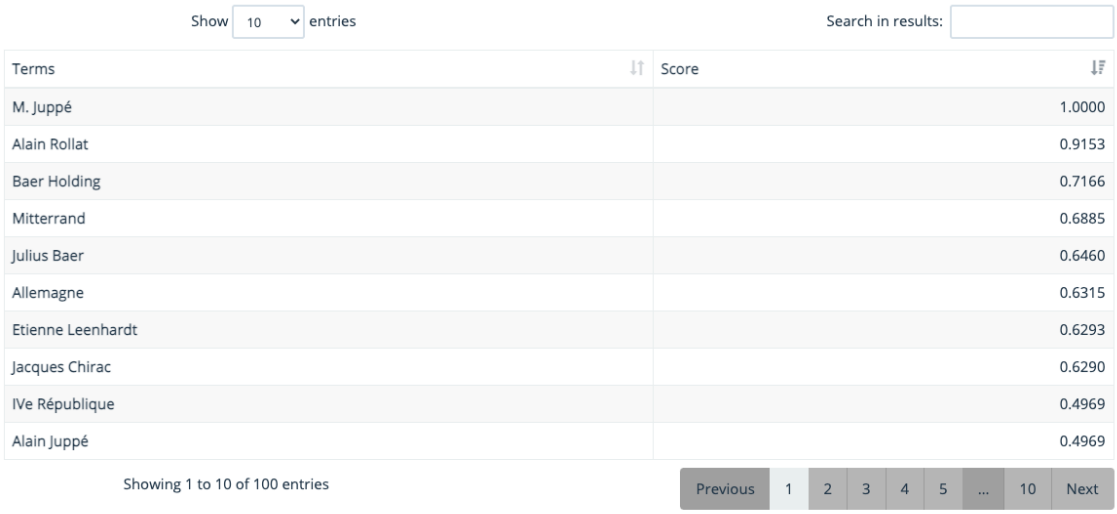
Using this interface as a debugging tool is also a way to use EVELIN.

From Figure 5.6, one can observe that the terms returned are actually entities not identified as such. Thus, even though the functionality cannot be used to learn the surrounding terms, it raises an issue in entity detection in the corpus.

In more general ways, EVELIN can be used to verify that entity disambiguation works well: Figure 5.8 shows that "Jacques Chirac" has been identified correctly, as the surrounding terms, locations and actors are relevant to the French politician. The subgraph tool is an efficient way to detect anomalies in entity detection.

5.3.3 TO GO FURTHER

Linking the results of the EVELIN interface to the Impresso-LOAD interface would create an interactive experience for users: once the graph has been browsed and information has been learned, a user could want to see the related articles he or she saw. Just like the original LOAD graph on Wikipedia data, that linked entities to their Wikipedia page, and sentences and pages to their Wikipedia link, the Impresso-EVELIN interface could link the entities to their Wikipedia page, and the sentences and articles to their related article or newspaper issue from the Impresso interface.

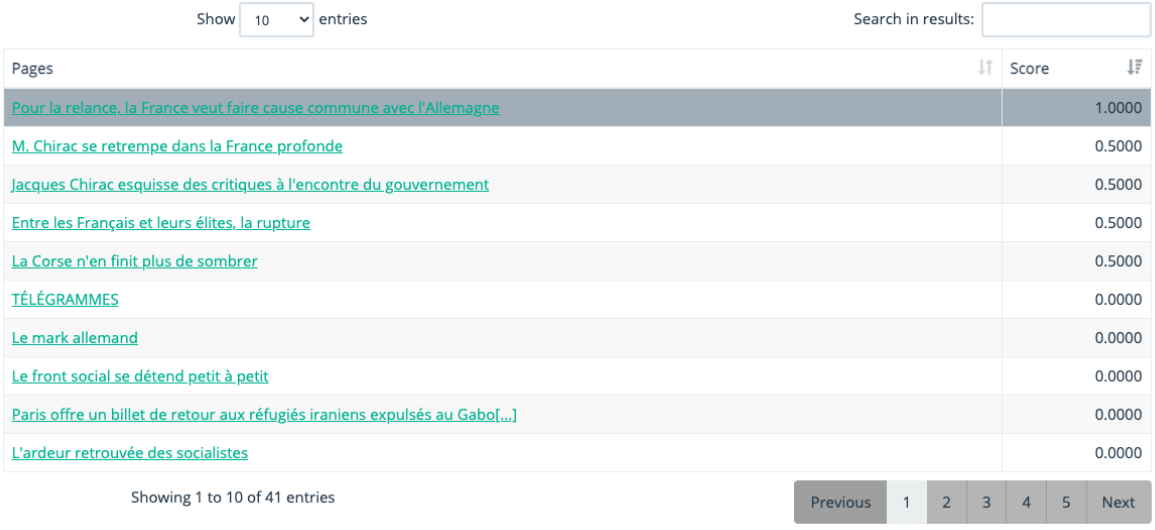


Show 10 entries Search in results:

Terms	Score
M. Juppé	1.0000
Alain Rollat	0.9153
Baer Holding	0.7166
Mitterrand	0.6885
Julius Baer	0.6460
Allemagne	0.6315
Etienne Leenhardt	0.6293
Jacques Chirac	0.6290
IVe République	0.4969
Alain Juppé	0.4969

Showing 1 to 10 of 100 entries Previous 1 2 3 4 5 ... 10 Next

FIGURE 5.6
EVELIN page result when querying terms related to actor entity Jacques Chirac



Show 10 entries Search in results:

Pages	Score
Pour la relance, la France veut faire cause commune avec l'Allemagne	1.0000
M. Chirac se retrempe dans la France profonde	0.5000
Jacques Chirac esquisse des critiques à l'encontre du gouvernement	0.5000
Entre les Français et leurs élites, la rupture	0.5000
La Corse n'en finit plus de sombrer	0.5000
TÉLÉGRAMMES	0.0000
Le mark allemand	0.0000
Le front social se détend petit à petit	0.0000
Paris offre un billet de retour aux réfugiés iraniens expulsés au Gabo[...]	0.0000
L'ardeur retrouvée des socialistes	0.0000

Showing 1 to 10 of 41 entries Previous 1 2 3 4 5 Next

FIGURE 5.7
EVELIN page result when querying articles related to actor entity Jacques Chirac

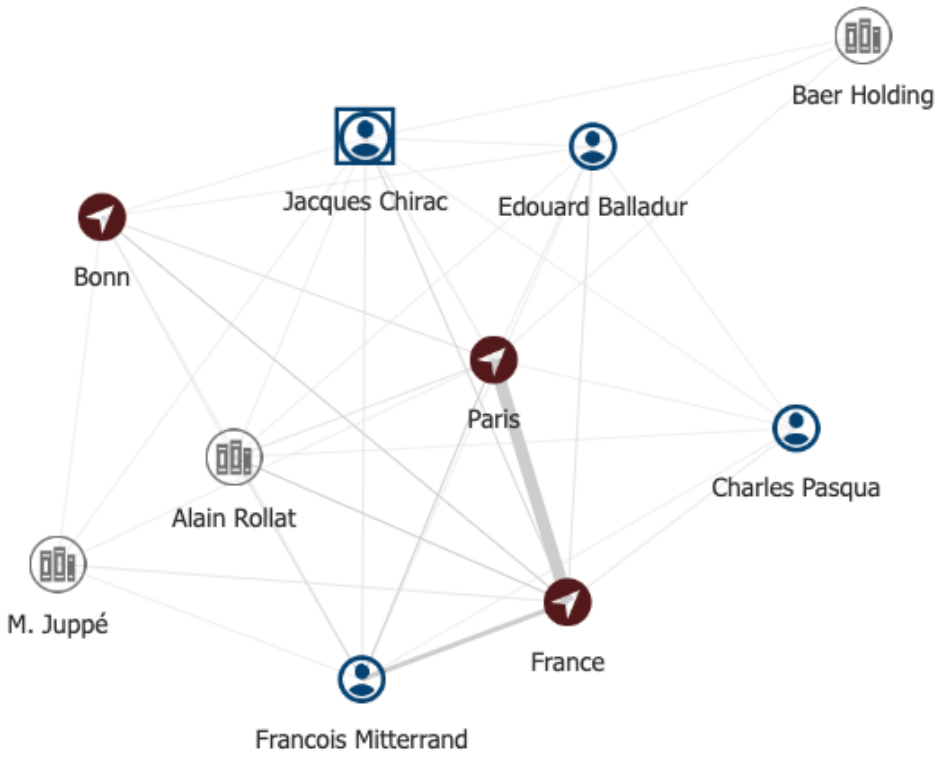


FIGURE 5.8
EVELIN page result when querying a subgraph related to actor entity Jacques Chirac

CHAPTER 6

CONCLUSION

6.1 CONTRIBUTIONS SUMMARY

The main contributions to the project were the following:

- Create a documentation and a summary for the impresso-LOAD graph creator.
- Render the code more modular and create prompts helping understanding steps of the code and debugging
- Render the code compatible with the EVELIN interface
- Make the code runnable on the cluster
- Make statistical and runtime analysis on the data and the code to understand the runtime problems.

6.2 FUTURE WORK

This section lists the work that should be done to further this project.

- **Non-artificial sentences**: As mentioned in Chapter 2.1.3, sentences in the Impresso-LOAD network are represented by 7 consecutive tokens: these are artificial sentences. One improvement would be to improve the quality of the sentence segmentation on noisy OCR data.
- **Statistical analysis of the corpus**: as mentioned in Chapter 4.2, runtime is currently an issue with the code. A statistical analysis on the corpus, to obtain information that influences the creation of the graph, such as the distribution of entities on the corpus, could help in figuring out how to render the code more efficient.
- **Performance improvement**: The code is robust to generate a graph on the entire corpus. Language management, memory management has been handled, and thus the full graph could be generated. Of course, this should not be done before the runtimes improvement have been realized.
- **Database adaptation**: for a full integration of LOAD/EVELIN into impresso, replacing the MongoDB backend with a MySQL database would avoid the multiplication of backend systems (MySQL, MongoDB, S3, Solr).

BIBLIOGRAPHY

[1] EPFL-DHLAB, ICL-UZH and C2DH. *impresso project*. URL: https://impresso-project.ch/.

[2] Andreas Spitz and Michael Gertz. *Terms over LOAD: Leveraging Named Entities for Cross-Document Extraction and Summarization of Events*. New York, NY, United States: Association for Computing Machinery, 2016.

[3] Isabelle Pumford et al. *impressoLOAD: Creating a Knowledge Graph Impresso Corpus*. Tech. rep. Canton de Vaud, EPFL, 2020.

[4] Andreas Spitz, Satya Almasian and Michael Gertz. *EVELIN: Exploration of Event and Entity Links in Implicit Networks*. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2017.

[5] Jakub Piskorski. *ExPRESS – Extraction Pattern Recognition Engine and Specification Suite*. Potsdam, Germany, 2007.

[6] D.B. Nguyen et al. *AIDA-light: High-throughput named-entity disambiguation*. Seoul, South Korea, 2014.

[7] Andreas Spitz et al. *Implicit Entity Networks: A Versatile Document Model*. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2020.