POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioningenieria

Graph Summarization on Linked Open Data

Advisor: Prof. Marco Colombetti

Alejandro Onatra, matricola 763773

Academic year 2016-2017

To my family that above and beyond is the best thing that I have in my life.

Abstract

One of the biggest challenges in the area of intelligent information management is the exploitation of the Web as a platform for data and information integration as well as for search and querying. The Linked Data paradigm has evolved as a powerful enabler for the transition of the current document-oriented Web into a Web of interlinked Data and, ultimately, into the Semantic Web. The most visible example of adoption and application of the Linked Data principles has been the **Linking Open Data Project**. Despite the wealth of information contained in the Web of Linked Data, ordinary Web users, not familiar with Semantic Web technologies and the specific application domains, are difficult to directly consume this information. In this research we study the summarization of graphs stored as Linked Open Data and its visualization, using heuristic and heterogeneous graph summarization algorithms. The purpose of the study was to understand the limitations and advantages of this algorithms in the summarization process and to the implement necessary changes to them in order to overcome this limitations creating a comparison chart and to obtain possible direction to guide future research on this topic. The selected set of algorithms were based on the Snap algorithm, these, are a group of graph summarization algorithms that can work on heterogeneous networks. In order to perform the observations we develop a testing platform based on the LDVM (Linked Data Visualization Model) with a *Client-Server* architecture. From the initial phase of the study the LOD problems when summarizing a graph were partially solved modifying using methods like *Hub* and compressing the graph as a post-processing practice. Within the set of modified *Snap family* of algorithms used in this work, the one that performed the best according to *execution time and grouping number* was the compressed, combined *kSnap, Hub* one. The advantages of this algorithm was the initial big grouping, contained iterative cycles for creating new groupings and cleaner output.

Sommario

Una delle sfide più importanti nella gestione intelligente delle informazioni è costituita dallo sfruttamento del Web come piattaforma per l'integrazione di dati e informazioni, nonché per la ricerca e la risposta a query. Il paradigma dei linked data si è evoluto come un potente strumento per la transizione dal Web orientato ai documenti a un Web di dati interconnessi e, infine, al Web semantico. L'esempio più significativo di applicazione dei principi dei linked data è stato il Linking Open Data Project. Nonostante la ricchezza delle informazioni contenute nel Web deilinked data, gli utenti ordinari delWeb hanno difficoltà a utilizzare direttamente questi dati a causa della loro scarsa familiarità con le tecnologie del Web Semantico. La mia ricerca si occupa della sintesi dei grafici memorizzati come Linked Open Data e della loro visualizzazione, utilizzando algoritmi euristici e eterogenei di sommarizzazione di grafi. Lo scopo dello studio era comprendere i limiti e i vantaggi di questi algoritmi nel processo di sommarizzazione e implementare le modifiche necessarie per superare le limitazioni, operando un confronto sistematico in vista di una possibile direzione futura di ricerca in questo ambito. Il set di algoritmi selezionato si basava sull'algoritmo Snap: si tratta di un gruppo di algoritmi di sommarizzazione di grafi che possono funzionare su reti eterogenee. Per eseguire le nostre analisi abbiamo sviluppato una piattaforma di test basata sul modello LDVM (Linked Data Visualization Model) con un'architettura client-server. I problemi di sommarizzazione di grafi di Linked Open Data sono stati parzialmente risolti utilizzando metodi come Hub e comprimendo il grafo ottenuto in fase di post-elaborazione. Nell'ambito degli algoritmi Snap modificati utilizzati in questo lavoro, quello che si è comportato meglio in base al tempo di esecuzione e al numero di raggruppamenti è stato una versione di kSnap. I vantaggi di questo algoritmo sono il grande raggruppamento iniziale, i cicli iterativi per la creazione di nuovi raggruppamenti e i risultati più trasparenti.

Acknowledgements

This work is dedicated to my parents that support me in the best way possible in order to finish my academic life. My sister that have been always an inspiration, keeping me motivated, motivation that help me finish this work. Finally the friends that in spite of being ignored many weekends support me and keep me company in the few moments that I had free time to share with them. Many people contribute to this work directly or indirectly, to those who I didn't mentioned, than you!

Contents

List of Figures

Chapter 1

Introduction

In this chapter we will introduce the work done in this research and an overview of the organization of this document. The first section presents the general motivations of the study. The second one present the problem that this work is solving. In the third section will present the scope of the work, the fourth one will define the most important terms and acronyms. Finally an overview of the document chapter will be presented.

1.1 Statement of the problem

Many data sets of interest today are best described as a linked collection of interrelated objects. These may represent homogeneous networks, in which there is a single-object type and link type, or richer, heterogeneous networks, in which there may be multiple object and link types (and possibly other semantic information). Examples of homogeneous networks include: single mode social networks, such as people connected by friendship links, or the WWW, a collection of linked web pages. Examples of heterogeneous networks include those in medical domains describing patients, diseases, treatments and contacts. Most of this information is stored as in an interconnected network of data silos connected through the internet and may or may not be available for the general public.

One of the biggest challenges in the area of intelligent information management is the exploitation of the Web as a platform for data and information integration as well as for search and querying. The Linked Data paradigm has evolved as a powerful enabler for the transition of the cur-

rent document-oriented Web into a Web of interlinked Data and, ultimately, into the Semantic Web. Just as initially we published unstructured textual information on the Web as HTML pages and search such information by using keyword-based search engines, now we are already able to easily publish structured information, reliably interlink this information with other data published on the Web and search the resulting data space by using more expressive querying beyond simple keyword searches.

The most visible example of adoption and application of the Linked Data principles has been the **Linking Open Data Project** a grassroots community effort founded in January 2007 and supported by the W3C Semantic Web Education and Outreach Group. The original and ongoing aim of the project is to bootstrap the Web of Data by identifying existing data sets that are available under open licenses, converting these to RDF according to the Linked Data principles, and publishing them on the Web.

Imagine hundred thousands of nodes , holding the relationships and attributes of individuals, one such type of networks are the social networks. With the size of this networks, is difficult to grasp some of the underlying implicit and explicit structures. There is an overwhelming wealth of information encoded in these graphs, there is a critical need for tools to summarize large graph data sets into concise forms that can be easily understood. Graph summarization has attracted a lot of interest from a variety of research communities, including sociology, physics, and computer science. It is a very broad research area that covers many topics. Different ways of summarizing and understanding graphs have been invented across these different research communities. These different summarization approaches extract graph characteristics from different perspectives and are often complementary to each other.

Despite the wealth of information contained in the Web of Linked Data, ordinary Web users, not familiar with Semantic Web technologies and the specific application domains, are difficult to directly consume this information resource. Linked Data visualization can alleviate this problem, visualization can be a reasonable way to visually present the internal structure in the data and the relationship between the data; friendly visualization interfaces allow the users to identify any unreasonable, incorrect or duplicate data and links in the Linked Data, thus helping the users intuitively and efficiently analyze data.

Currently *ordinary* web users and experts can benefit from enhanced *vi-*

sualization methods that can aid them to understand implicit and explicit information contained in the Linked Data. One such technique is *graph summarization* that is at the core of aiding the user to obtain relevant information from a graph by reducing the data space and presenting only the most important data. One such data sets are the ones belonging to the *Linked Open Data* movement. These sets are used as base to diverse applications, one such is the Google's *Knowledge Graph*.

In order to better understand how summarization works in *Linked Open Data*, a set of this kind of algorithms was chosen to *test their behaviour and define limitations and possible enhancements that could be implemented*. The algorithm set are based on the Snap algorithm, these, are a group of graph summarization algorithms that can work on heterogeneous networks and unlike others, focus on summarizing based on attributes and relationships. One of the advantages of this set is the flexibility to implement them.

1.2 Scope and delimitation

The research presented in this document study the summarization of graphs stored as Linked Open Data and its visualization, using heuristic and heterogeneous graph summarization algorithms. In order to do this the research focuses in the areas of $LinkMining$, $LinkData$ and $GraphVisualization$. The purpose of the study is to understand the limitations and advantages of this algorithms in the summarization process and to implement necessary changes to them in order to overcome this limitations creating a comparison chart and to obtain possible direction to guide future research on this topic.

1.3 Methodology

This study uses an exploratory approach in order to study the phenomenon described on Section 1.2. Therefore there is no initial formal hypothesis on how the summarization will behave on Linked Open Data. There are informal hypothesis on the behaviour based on the conclusions from previous studies on the subject.

In order to perform the observations it was necessary to have an experimental setup. Up to the best of our knowledge there was no suitable experimental platform at the moment that this work started so it was necessary

to create the *GSummarizer* experimental platform as part of the research.

The original algorithms were modified initially to adjust to multiple relation and multiple attribute handling. The test were performed as a first phase of the study. After the initial assessment, a set of changes and possible optimizations were determined. For the second phase of the studies, a new set of optimized algorithms were implemented to be tested and obtained the final comparative results.

1.4 Significance of the study

The study of the Snap-based algorithms applied to LOD data sets will aid the research done on graph visualization and graph summarization using heuristic clustering algorithms. The research field will benefit of the insights obtained from this research because it can direct future work towards better graph summarization techniques and reuse the testing platform implemented.

1.5 Document organization

The body of this document is divided in six chapters starting with the current one that introduce the reader to the overview of the research described in this document. The chapters are organized as following:

- **Chapter 2:** This chapter presents the literature review and the state of the art on the topics of this work.
- **Chapter 3:** The third chapter presents the original algorithms, the initial modifications and the final optimizations.
- **Chapter 4:** Presents the testing platform, *GSummarizer* motivations, design decisions and architectural details.
- **Chapter 5:** The fifht chapter describes the experimental procedures and the experimental analysis and results.
- **Chapter 6:** This chapter presents the conclusions and future work sections.

1.6 Acronyms

- **LOD** Linked Open Data
- **LM** Linked Mining
- **DM** Data Mining
- **GS** Graph Summarizer
- **LDVM** Linked Data Visualization Model

Chapter 2

Literature Review

This chapter describes chronologically how the literature was reviewed, compiled an analyzed in order to reach the critical to start the main researching activity. As is expected this chapter will start by the bigger concept in which this research is supported: **Linked Open Data**, from here, the literature path topics will get more specific until reach the specific topic of interest: **Graph Summarization in Linked Open Data**

This chapter discusses the literature as followed by us, covering the path that we took to gather all the support information needed to start the main experimental activities. The main sections of this chapter are ordered as follows, the first section will talk about the entry topic in owr research: *Linked Open Data (LOD)*, here we will start talking about Linked Data, from the conceptual point of view and then we will discuss about the leading technologies implementing it, particularly focusing in the most common one, the *Resource Description Framework (RDF)*. The second section talks about the concepts and state of the art of *Data Mining (DM)* and the problems that arises when applied to LD, the third section will talk about, *Link Mining (LM)* concepts and techniques and how LM solve some of the issues that traditional DM techniques have when working with LD. Finally we will connect the current state of the research with the main topic of this document: *Graph Summarizing (GSum)*, the main concepts of this topic and how it is related to LM and how it is applied in general to LOD.

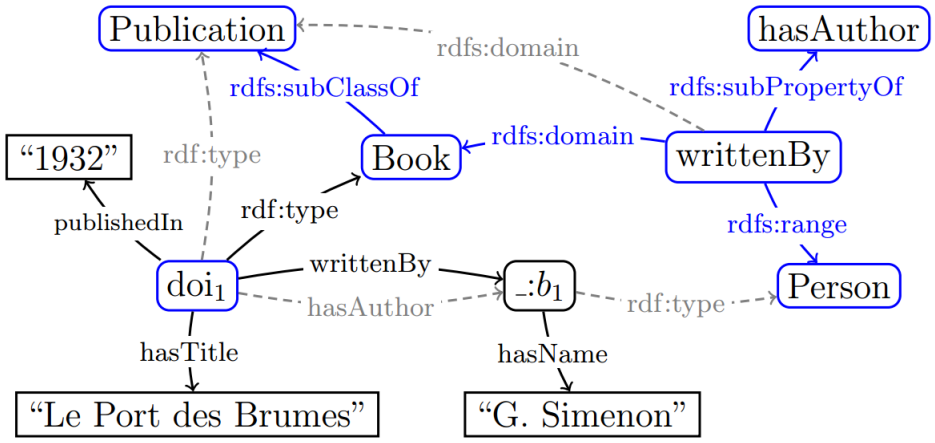
2.1 Link Open Data

Many data sets of interest today are best described as a linked collection of interrelated objects. These may represent homogeneous networks, in which there is a single-object type and link type, or richer, heterogeneous networks, in which there may be multiple object and link types (and possibly other semantic information). Examples of homogeneous networks include: single mode social networks, such as people connected by friendship links, or the WWW, a collection of linked web pages. Examples of heterogeneous networks include those in medical domains describing patients, diseases, treatments and contacts [15]. One of this networks is the World Wide Web, this one, has radically altered the way we share knowledge by making easier to publish data with high accessibility. From the human accessibility point of view *hyperlinks* allow users to traverse the data space (Websites, Media, etc), *Search engines* exploit the structure of the World Wide Web to find relevant documents for the users, inferring the user preference based on the information and analyzing the link structure in them [41]. But for many years, data published in the Web was available as raw dumps in formats as CSV, XML or as HTML tables. The problem with this is that information was lost over the lack of expressive power either from the previous formats or HTML itself, this means that we were losing *the description of individual entities, the relations and the type of relations among them* [3]. However, in recent years the Web has evolved from a global information space of linked documents to one where both documents and data are linked. One of the advantages of being able to relate information with each other is that a machine can understand explicitly the relation between data, between concepts. Underpinning this evolution is a set of best practices for publishing and connecting structured data on the Web known as Linked Data. The adoption of the Linked Data best practices has lead to the extension of the Web with a global data space connecting data from diverse domains such as people, companies, books, scientific publications, films, music, television and radio programmes, genes, proteins, drugs and clinical trials, on-line communities, statistical and scientific data, and reviews. This Web of Data enables new types of applications [3]. One of the most successful projects coming from the Link Data paradigm is the Link Open Data projects, used as an open Link Data repository where continuously data is being pushed either by automatic programs, privates or public entities.

2.1.1 The web of data

Data can be expressed in many ways and one of the issues that systems have in order to communicate among each other is that they don't speak the same language, they don't see and define the world with the same model. A solution to this problem is the use of *Application Programming Interface (APIs)* , due to the many ways APIs are typically designed application development based on Web 2.0 mashups is often burdensome and doesn't scale well. In addition to requiring effort to repeatedly learn new interfaces, the so-called created data is locked in its respective platform.[19]. Therefore, APIs work as a middleware solution enforcing a contract between systems in order for the communication to go as if the involved systems speak the same language, still it doesnt scale well as if the two systems could speak in the same language, or similarly enough. In order to two systems to speak in the same language, they must define the world in a similar way, in order to do that, their model have to be similar and their description language compatible or the same. The RDF Vocabulary Definition Language (RDFS) [45] and the Web Ontology Language (OWL) [40] provide a basis for creating vocabularies that can be used to describe entities in the world and how they are related. Vocabularies are collections of classes and properties. *RDF Schema* allows enhancing the descriptions in RDF graphs by means of RDFS triples, declaring semantic constraints between the graph classes and properties. The RDFS constraints shown in Figure 2.1 lead to implicit triples which mabe part of an RDF graph even though they are not physically present in it. An implicit triple can be obtained by an immediate entailment step based on (i) an RDFS constraint, and (ii) either a second constraint (also called schema triple) or an RDF triple that is not a constraint (also termed data triple).

Vocabularies are themselves expressed in RDF, using terms from RDFS and OWL, which provide varying degrees of expressivity in modelling domains of interest. Anyone is free to publish vocabularies to the Web of Data [2], which in turn can be connected by RDF triples that link classes and properties in one vocabulary to those in another, thereby defining mappings between related vocabularies. By employing HTTP URIs to identify resources, the HTTP protocol as retrieval mechanism, and the RDF data model to represent resource descriptions, Linked Data directly builds on the general architecture of the Web [57]. The Web of Data can therefore be seen as an additional layer that is tightly interwoven with the classic document



$$\mathtt{G} = \{\mathtt{doi_1}\ \text{rdf:type}\ \mathtt{Book},\ \mathtt{doi_1}\ \mathtt{writtenBy}\ \_{:}b_1,\ \mathtt{doi_1}\ \mathtt{hasTitle}\ \text{“}\mathtt{Port\ des\ Brumes}\text{”},\ \_{:}b_1\ \mathtt{hasName}\ \text{“}\mathtt{G.\ Simenon}\text{”},\ \mathtt{doi_1}\ \mathtt{publishedIn}\ \text{“}\mathtt{1932}\text{”}\}$$

Figure 2.1: Sample RDF graph [58]

Web and has many of the same properties:

- The Web of Data is generic and can contain any type of data.
- Anyone can publish data to the Web of Data.
- Data publishers are not constrained in choice of vocabularies with which to represent data.
- Entities are connected by RDF links, creating a global data graph that spans data sources and enables the discovery of new data sources.

From an application development perspective the Web of Data has the following characteristics:

- Data is strictly separated from formatting and presentational aspects.
- Data is self-describing. If an application consuming Linked Data encounters data described with an unfamiliar vocabulary, the application can dereference the URIs that identify vocabulary terms in order to find their definition.

- The use of HTTP as a standardized data access mechanism and RDF as a standardized data model simplifies data access compared to Web APIs, which rely on heterogeneous data models and access interfaces.
- The Web of Data is open, meaning that applications do not have to be implemented against a fixed set of data sources, but can discover new data sources at run-time by following RDF links.

2.1.2 Linked data

One of the biggest challenges in the area of intelligent information management is the exploitation of the Web as a platform for data and information integration as well as for search and querying. Just as initially we published unstructured textual information on the Web as HTML pages and search such information by using keyword-based search engines, now we are already able to easily publish structured information, reliably interlink this information with other data published on the Web and search the resulting data space by using more expressive querying beyond simple keyword searches.

The Linked Data paradigm has evolved as a powerful enabler for the transition of the current document-oriented Web into a Web of interlinked Data and, ultimately, into the Semantic Web. The term *Linked Data* here refers to a set of best practices for publishing and connecting structured data on the Web [37]. In summary, Linked Data is simply about using the Web to create typed links between data from different sources. These may be as diverse as databases maintained by two organisations in different geographical locations, or simply heterogeneous systems within one organisation that, historically, have not easily being able to interoperate at the data level. Technically, Linked Data refers to data published on the Web in such a way that it is machine-readable, its meaning is explicitly defined, it is linked to other external data sets, and can in turn be linked to from external data sets. [3].

Linked Data relies on two technologies that are fundamental to the Web: *Uniform Resource Identifiers* (URIs) [46] and the *HyperText Transfer Protocol* (HTTP). While *Uniform Resource Locators* (URLs) have become familiar as addresses for documents and other entities that can be located on the Web, Uniform Resource Identifiers provide a more generic means to identify any entity that exists in the world. Where entities are identified by URIs that use the **http://** prefix scheme, these entities can be looked up simply by dereferencing the URI over the HTTP protocol. In this way, the HTTP pro-

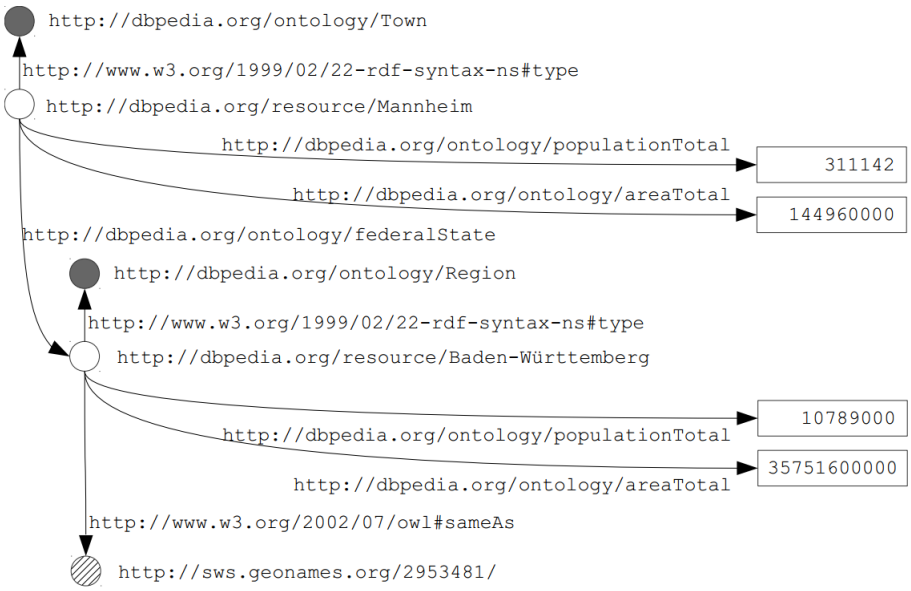


Figure 2.2: RDF diagram depicting the subject - relation - object between the nodes

tocol provides a simple yet universal mechanism for retrieving resources that can be serialised as a stream of bytes (ie. The picture of a person). Whilst HTML provides a means to structure and link documents on the Web, RDF provides a generic, graph-based data model with which to structure and link data that describes things in the world.

The RDF model encodes data in the form of **subject**, **predicate**, **object** triples. The subject and object of a triple are both URIs that each identify a resource, or a URI and a string literal respectively. The predicate specifies how the subject and object are related, and is also represented by a URI [3]. For example, an RDF triple can state that a person *Berners* is a member of the *DIG* department, the subject, object and predicate, each identified by an URI. Similarly an RDF triple may relate two films using the predicate *sameAs* in order to declare that the two URIs are referencing the same film. Another example is presented in Figure 2.2

While the primary units of the hypertext Web are HTML (HyperText Markup Language) documents connected by untyped hyperlinks, Linked Data relies on documents containing data in RDF (Resource Description Framework) format [44]. However, rather than simply connecting these documents, Linked Data uses RDF to make typed statements that link arbitrary things in the world. The result, which we will refer to as the Web of Data,

may more accurately be described as a web of things in the world, described by data on the Web. [48] outlined a set of 'rules' for publishing data on the Web in a way that all published data becomes part of a single global data space:

1. Use URIs as names for things.
2. Use HTTP URIs so that people can look up those names.
3. When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL).
4. Include links to other URIs, so that they can discover more things.

These have become known as the **Linked Data principles**, and provide a basic recipe for publishing and connecting data using the infrastructure of the Web while adhering to its architecture and standards.

2.1.3 Linked data technology stack

Linked Data [1, 21] denotes the emerging trend of publishing structured data on the Web as interlinked RDF graphs [29] following a number of simple principles: use HTTP URIs to name things, return RDF about those things when their URIs are looked up, and include links to related RDF documents elsewhere on the Web. These trends — and related trends with respect to embedding metadata into HTML pages (such as promoted by schema.org) have lead to a huge volume of RDF data being made openly available online, contributing top the *Web of Data*.

The Semantic Web provides the necessary tools to query this data, **(i)** firstly by defining RDF [29, 18] as a universal data format; **(ii)** secondly by defining SPARQL [51, 52], a standard query language for RDF; and **(iii)** lastly by providing schema languages such as RDF Schema (RDFS) [45] and OWL [39], which allow for adding rich semantics to the data. These three components, and in particular the reasoning capabilities enabled by RDFS and OWL, are essential to enable usage of the Web of Data as *one huge database* as originally envisioned by *Tim Berners-Lee* [56].

One of the goals of having a Web of Data is allow "machines" to infer or process from different sources of information describing the world in a common *language*, and further more, reason and create statements on top

of it, but with the current technologies applied to the web of data and as opposed to standard reasoning and query answering in OWL reasoning over Linked Data poses several unique challenges [27]:

1. Linked Data is huge, that is, it needs highly scalable or modular reasoning techniques.

2. Linked Data is not "pure" OWL, that is, a lot of RDF Data published as Linked Data violates the strict syntactic corset of OWL DL, and thus is not directly interpretable under OWL Direct Semantics.

3. Linked Data is inconsistent, that is, if you take the Web of Data in its entirety, it is quite normal to encounter inconsistencies - not only from accidental or malicious datasets but also because publishers may express contradicting views.

4. Linked Data is evolving, that is, RDF Graphs on the Web evolve, they change, information is added and removed.

5. Linked Data needs more than RDFS and OWL, that is, there is more implicit data hidden in Linked Data than can be captured with the semantics of RDFS and OWL alone.

The Resource Description Framework — RDF

Informally, all RDF data can be understood as a set of subject—predicate—object triples, where all subjects and predicates are URIs, and in the object position both URIs and literal values (such as numbers, strings, etc.) are allowed. Furthermore, blank nodes can be used in the subject or object resource to denote an unnamed resource with local scope. More formally, given the set of URI references U, the set of blank nodes B, and the set of literals L, the set of RDF constants is denoted by $C := U \cup B \cup L$ A triple $t := (s, p, o) \in UB \times U \times C$ is called an RDF triple, where s is called subject, p predicate, and o object. A triple $t := (s, p, o) \in Tr, Tr := C \times C \times C$ is called a generalised triple [25], which allows any RDF constant in any triple position: henceforth, we assume generalised triples unless explicitly stated otherwise. We call a finite set of triples $G \subset Tr$ a graph. In the Turtle syntax, URIs are often denoted either in full form using strings delimited by angle brackets (e.g. $< http : //dbpedia.org/resource/Werner\_von\_Siemens >$)

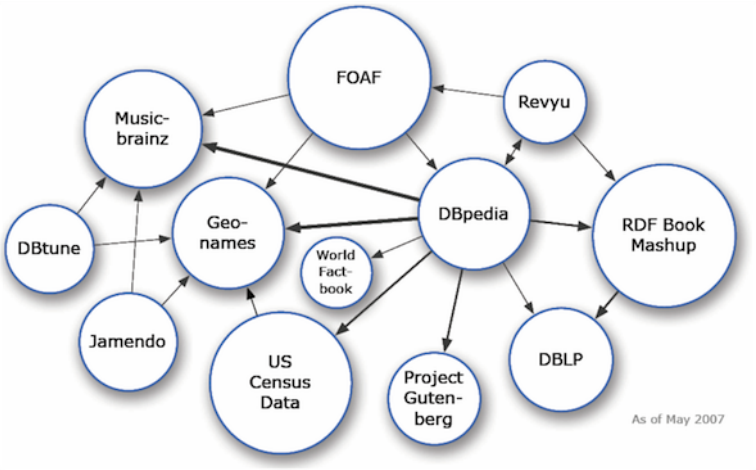


Figure 2.3: Linked Open Data Graph in 2007

or as shorthand prefixed names (e.g. dbr:Werner_von_Siemens). Literals are delimited by double quotes (e.g. "SAP" "AG") with an optional trailing language tag (e.g. @en, @de) or a data type which itself is again identified by a URI (e.g. ^^xsd:dateTime). Blank nodes are (typically) scoped to a local document and are denoted in Turtle either by the "prefix" _: or alternatively using square brackets "[", "]". Furthermore, Turtle allows use of "a" as a shortcut for **rdf:type** (denoting class membership) [5]. Finally, RDF triples in Turtle notation are delimited by a trailing *'.'*, where predicate-object-pairs that share the same subject can be grouped using *';'* and several objects for the same predicate can be grouped using *','* [42]

2.1.4 Linked Open Data

The most visible example of adoption and application of the Linked Data principles has been the **Linking Open Data Project** [53], a grassroots community effort founded in January 2007 (Figure 2.3) and supported by the W3C Semantic Web Education and Outreach Group. The original and ongoing aim of the project is to bootstrap the Web of Data by identifying existing data sets that are available under open licenses, converting these to RDF according to the Linked Data principles, and publishing them on the Web.

Participants in the early stages of the project were primarily researchers and developers in university research labs and small companies. Since that

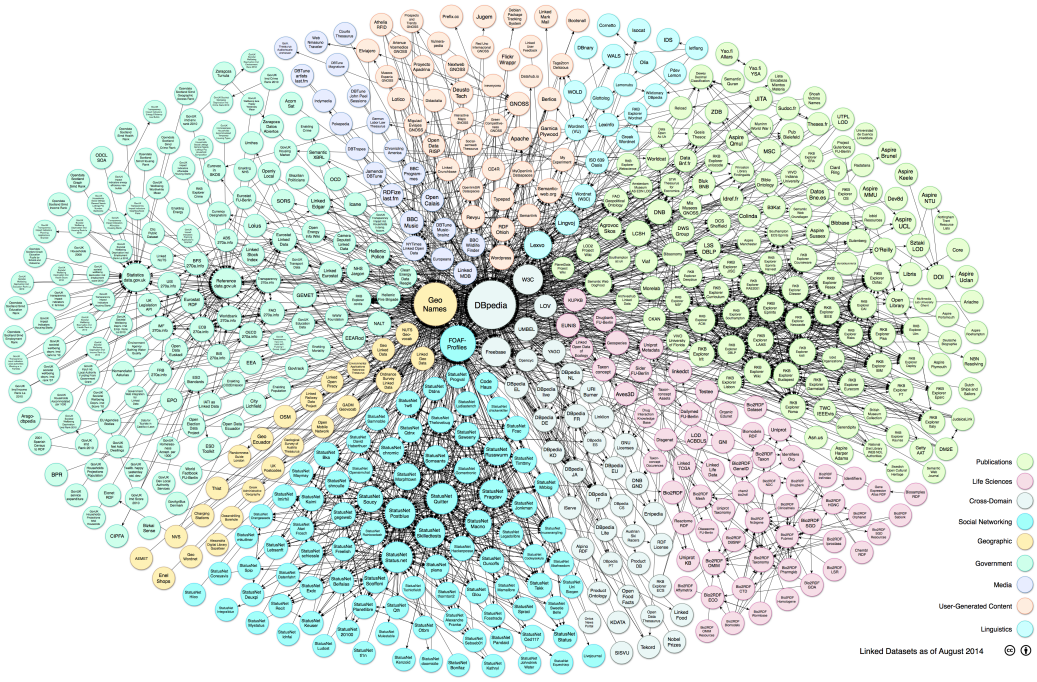


Figure 2.4: Linked Open Data Graph in 2014

time the project has grown considerably, to include significant involvement from large organisations such as the BBC, Thomson Reuters and the Library of Congress. This growth is enabled by the open nature of the project, where anyone can participate simply by publishing a data set according to the Linked Data principles and interlinking it with existing data sets. An indication of the range and scale of the Web of Data originating from the Linking Open Data project is provided in Figure 2.4. Each node in this cloud diagram represents a distinct data set published as Linked Data.

The size of the Web of Data can be estimated based on the data set statistics that are collected by the LOD community, according to these statistics, the Web of Data currently consists of 4.7 billion RDF triples, which are interlinked by around 142 million RDF links (May 2009). As Figures 2.3, 2.4 and 2.5 shows, certain data sets serve as linking hubs in the Web of Data. For example, the **DBpedia** data set consists of RDF triples extracted from the "infoboxes" commonly seen on the right hand side of Wikipedia articles, while *http* : *//www.geonames.org/ontology/* provides RDF descriptions of millions of geographical locations worldwide. As these two data sets provide URIs and RDF descriptions for many common entities or concepts, they are

frequently referenced in other more specialised data sets and have therefore developed into hubs to which an increasing number of other data sets are connected.

The concept of LOD is based on the idea of linking publicly available data "silos" on the internet by means of semantic methods and networks. By linking data, all of the data objects (e.g., man, woman) become related to each another. By determining a number of rules about these relationships, such inter-linked data can be "understood" by machines and algorithms, which enables *global data mining approaches and the discovery of truly new associations, patterns and knowledge*. LOD is based on the Resource Description Framework (RDF) data model, which formulates syntax and rules about data and resources as well as their location on the internet. The RDF-model resembles classic methods for concepts such as the *EntityRelationship-Model (ERM)* or the *Unified Modelling Language (UML)* class diagram. RDF models have a formal semantic that is based on digraphs. The RDF syntax extends the Extensible Hyper Text Markup Language (XHTML) of the web by semantic annotation in support of linking up data. Along with RDF there are also other standards such as the Web Ontology Language (OWL) that enable more complex semantic annotations as discussed earlier in section 2.1.1. The LOD approach is not about expanding linkage of documents via hyperlinks something which has been common on the Internet but, rather, about the "linking of all publically accessible data" together [3]. Publically accessible data includes many areas of interest, such as weather, events, politics, business, news, geographic spatial data, videos, pictures, images and much more.

More and more people, institutions, companies and research centers are putting their data on the web in a machine readable form as per the LOD approach thereby making it available to the general public for complex and interdisciplinary data analysis. Google, for instance, uses LOD to improve search functions. DBpedia is an LOD version of the open and free online encyclopedia Wikipedia. [48]

2.2 Data Mining in Linked Open Data

Now that we understood *Linked Open Data (LOD)*, the *Resource Description Frameworks (RDF)* and how they are implemented currently, we will focus on how this technologies can be used to expand the knowledge that we have

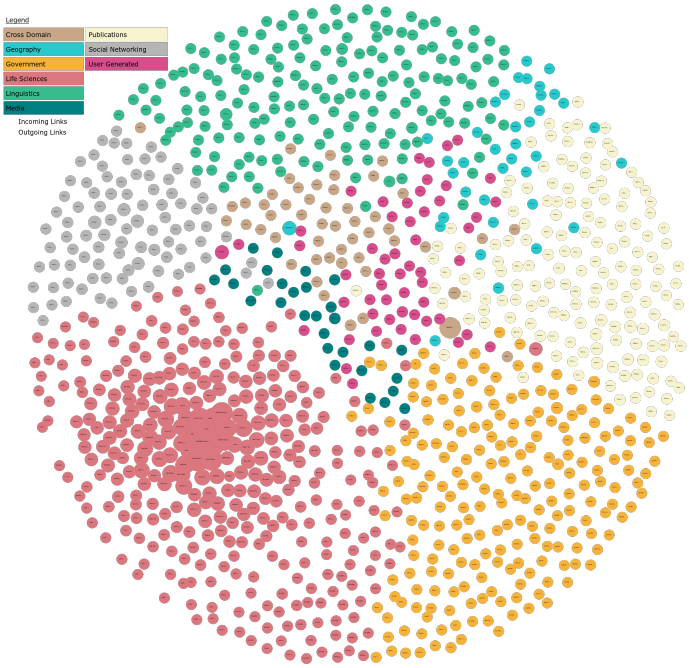


Figure 2.5: Linked Open Data Graph in 2017

regarding the data that they hold using data mining techniques.

Imagine hundred thousands of nodes , holding the relationships and attributes of individuals. There must be a way to better understand the information contained in that graph, in that DB. The information may not be explicitly available in plain sight, therefore we need a process to extract or better, to *infer* data from it. In order to do this we will start exploring how this is done for more traditional unstructured data sets and then move to the more specific are of *inference (INF) and knowledge extraction (KEX)* in an RDF based DB

The rapid development in information and computer technology has facilitated an extreme increase in the collection and storage of digital data. However, the associated rapid increase in digital data volumes does not automatically correlate with new insights and advances in our understanding of those data. The relatively new technique of data mining offers a promising way to extract knowledge and patterns from large, multidimensional and complex data sets. The rapid technological advances in information technology of the 21st century are intrinsically linked to a gigantic increase in data and information. This is the result of increased networking and globalization,

continuous improvement in computer engineering, media storage, highly sophisticated data bases, the Internet as a platform for communication and of the enormous expansion of automated data collection via sensors, monitoring systems and mobile and smartphone applications. The ever smaller-scale automated measuring, cataloging and monitoring of so many areas of people's lives as well as their social environment and networks leads to a "datafication" of the world refers to the phenomenon of growing data volume in all areas of life also as "information overload" [30]

With the increasing networking and ever-growing computing capacity we find ourselves in a "profound transformation process" of the 21st century where more and more "decision-making processes have to be outsourced to technical systems" in order to be at all able to function. Internet services like Facebook, Google and Wolfram Alpha aim "to make all systematic knowledge immediately computable and accessible to everyone". For the sake of comparison, "it would take over five years to watch the amount of videos which were transferred per second through the Internet in 2015" [30]. Whereas in the 1990s there was still a deficit in the availability of digital data, the relation between data availability and existing evaluation methods and algorithms has changed completely. Nowadays, generating data does not mean insight. Only a fraction of the data is used to gain insight. As always, the statement applies that "data is not insight". Those who have a lot of data "have possibilities to gain insight, but those who have the right analytical tools and instruments, also hold the key to acquiring insight". While large amounts of available data have, for the most part, been archived and stored, only small parts have really been analyzed, used and understood and processed in human-understandable ways [26]. In order to understand data and to gain insight, the data has to be sorted, transformed, harmonized and processed both statistically and analytically. The potential growth increase in data bases in all areas makes the analysis of huge amounts of more complex information more difficult and less clear. Using the procedures of classical statistics that have been common up to now, it is becoming apparent that very limited information and knowledge can be gleamed and insufficient progress made from the huge and complex amounts of data. Only if future research succeeds in extracting information from this complex amount of data can this be the basis for good operative and strategic decisions and projections. In recent years, an enormous development in the area of complex data analysis has been made with data mining. **Data mining**, often termed **knowledge**

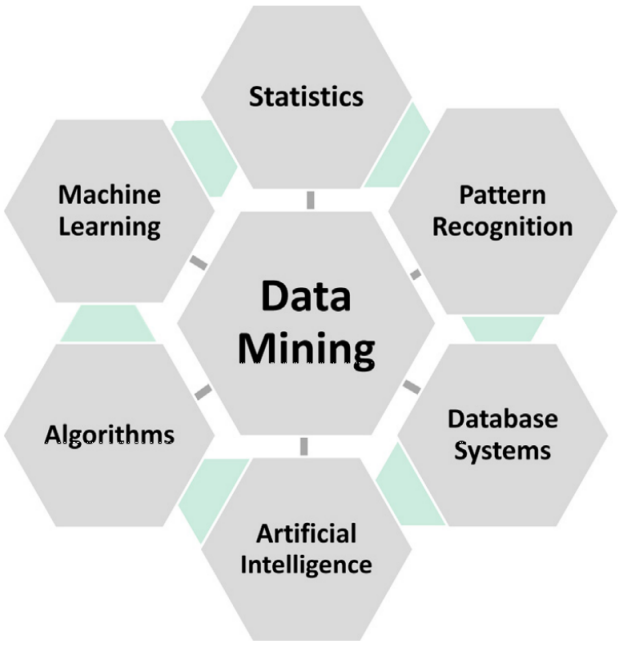


Figure 2.6: Confluence of different multiple disciplines in the data mining process.

discovery (KDD) in databases, is defined as the "non-trivial process of identifying valid, new and potentially useful and understandable patterns in data bases" [30] Data mining is a rather new field in computer science that *pursues the objective of extracting knowledge and analyzing complex data to find existing associations, to extract structures, patterns and regularities in large and complex data bases* [28]. Currently, there are a number of common data mining techniques to analyze texts, web contents, images, pictures, videos and spatial data. Compared to classical statistical techniques, all of these data mining techniques exhibit very good results when it comes to discovering and analyzing patterns and coherencies of the researched data.

Data mining involves the entire process from the provision of data right up to the projection and application of model findings to new, unknown data structures. This process includes:

- Techniques to preprocess data
- The actual data mining system
- Interpreting data

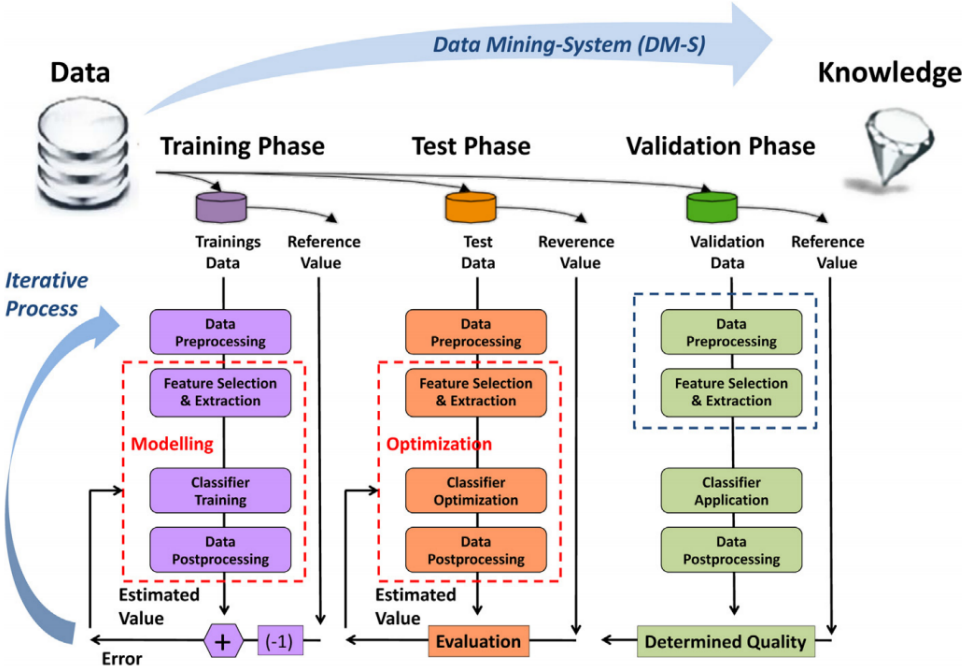


Figure 2.7: Data mining process with the data mining system (DM-S) in the Phases (1) Training Phase, (2) Test Phase and (3) Validation Phase. The data mining process works in a comparable way in all types of data mining types like text mining or web mining . Credits to [30].

- Evaluating data

Necessary *preprocessing* steps for data mining include data selection, preprocessing and transforming data into suitable data formats. The DM system itself is at the core of the actual data mining process, which is made up of three Phases: **Training**, **Test** and *Validation* as depicted in Figure 2.7. Within this process the objective of data mining is to repeatedly attempt to determine an estimated value (e.g., the buying behavior of a customer) based on the researched data (e.g., market data), which is compared with a predetermined reference value (target variable). This process is repeated iteratively until the comparison of estimated and reference values results in an acceptable value. This obtained model now forms the basis for Phase 3 of the interpretation and evaluation as well as for deriving knowledge based on other, as yet unknown data.

Data mining includes the analysis of numeric and categorical data in large

and complex data sets. Often this term is used to generally describe more specialized techniques, such as text, web or spatial data mining.

Text Mining: Text mining includes algorithms to analyze lexical and grammatical aspects of texts. In text mining algorithms the actual text is broken down into specific structures whereby patterns and core information of the text are recorded and grouped and classified using data mining methods. The first text mining tools could record contents and structures of simple text documents like Microsoft Word and Acrobat PDF documents. New developments in text mining now scan and analyze unstructured text in s, memos, surveys, chats, notes, white papers, forums and presentations [28]. **Web Mining:** Web mining is understood to be the application of data mining methods on information compiled on the Internet. In web mining, a distinction is made between (1) web content mining, which is the analysis of website contents; (2) web structure or relationship mining, i.e. the analysis of in-and outbound hyperlinks of websites and (3) web usage mining, which records and analyses user interaction with websites by scanning log files. **Image-Mining:** The goal of image mining techniques is to analyze and extract spatial patterns in image data which are not explicitly stored in the images [36]. Pattern extraction is done in a variety of ways based on recognizing the existence and distribution of colors, texture, shape, distances and intensities in the image data. **Spatial Data Mining (SDM):** Spatial data mining pursues the goal of discovering patterns in large, multi-dimensional spatial data sets as created by remote sensing techniques in earth observation. Extracting interesting patterns and associations from complex and multi-dimensional spatial data bearing spatial dependencies and spatial-temporal autocorrelations is more difficult than from traditional numeric and categorical data [49]. If, in addition to spatial data, additional time series are incorporated into the pattern analysis, the term spatial-temporal data mining is used to describe this.

There are various methods in support of data mining, which can be grouped according to the objective in question. Some of the most relevant methods are: **Association and sequence analysis:** With the help of association analysis, relationships between objects can be made and quantified. Using specific indicators, which, in most cases, include the support, confidence and lift values, the strength of the identified associations can be evaluated. *Grouping and clustering:* These are procedures which pool similar objects into groups with the goal of having very similar objects within

one group and having the groups differ as much as possible from each another. Numerous metrics are used for characterizing similarities in groups. *Regression:* With the assistance of regression analysis, functional dependencies are determined among the variables within a data set. Regression models are used to estimate or predict variables. To represent dependencies there are also linear and nonlinear (e.g., square, logistic or Poisson) regression approaches in addition to linear ones. **Classification:** The goal of classification is to find functions and models with whose help data objects can be assigned to previously identified classes. Deriving a model is based usually on a set of objects for which the respective class allocation is known. With the help of a model that has to be determined, objects are to be classified that have no known classification. The models can be determined with the help of *neuronal networks, discriminant analyses or decision treesand random forest. NaÃ¯ve Bayes methods as well as support vector machines (SVM) can also be used to classify data. There are a number of other procedures such as time series analysis or visualization and evolutionary algorithms.*

Linked Open Data (LOD) provide a way for exposing, sharing, and connecting data via dereferenceable URIs on the Web. There is an increasing number of interesting open data sets available on the Web, such as *DBPEDIA, GEONAMES, MusicBrainz, WORDNET, DBLP, etc.* published under various licenses. Dealing with knowledge bases in the Semantic Web poses various challenges due to problems of uncertainty and sparsity in the data, as well as issues with the scalability and rigidity of the processing mechanisms. Hence the necessity of data-driven methods arises also in this context. So far most of these tasks are tackled by borrowing and adapting **text mining** techniques among others. Structured knowledge bases in the Semantic Web need specific methods that are suitable for the specific data models employed in this context [13]. Some of the biggest challenges while using Knowledge discovery techniques are the following:

Sparse and Missing Data: Dealing with Semantic Web knowledge bases faces various obstacles related to the very large scale and nature of such data. As the worst case complexity of the algorithms for the basic reasoning tasks seems to indicate that they are unfeasible for application to large scale knowledge bases, especially when expressed in rich standard representations. However, many reasoning engines demonstrate good scalability and performance with a constant effort towards optimization (or approximation). The most important problem when learning from LOD is data sparseness.

An abundance of *missing data is due to the inherent open nature of the Semantic Web* which makes each attempt to circumscribe knowledge a failure. Makinhe closed-world assumption (as in the context of databases) may (partially) help coping with data sparseness but such an assumption would also affect the intended semantics of the data. The same would happen adopting alternative forms of (auto-)epistemic inference. When querying the LOD cloud, one is interested to statements (implicit triples) that may be derived by some form of (deductive but also inductive) inference rather than those that are already explicitly contained in the knowledge base. Hence, generally some form of inferred closure is sought, i.e. the completion of the knowledge base with (all the) statements that (definitively or likely) hold according to the intended semantics. **Complexity and Efficiency** There is also an issue with scalability of the underlying learning processes in terms of the complexity of the datasets in terms of both size and representation given the graph structure that can escalate quickly and render the methods quite complex [13].

Semantic Web Mining constitutes a sub-field of Data Mining and Semantic Web research that spans nearly over the last decade. Initially it has been targeting the semantics hidden in the documents published on the Web with techniques borrowed from text mining. With the advent of the *Web of data*, new efforts have focused on **mining ontologies** in the Semantic Web. Adaptations of statistical relational learning methods to graphs or other standard representations backing the ontologies in the Semantic Web have provided new techniques for mining structured knowledge bases. In [13] the authors expand more in detail the more mature techniques in this area. There is also another subfield that works not directly with the ontology's rather than with the graph itself, it is the field of *Link Mining* that will be discussed in the next section.

2.3 Link Mining

When working with RDF graphs, in order to extract the knowledge implicit in it, we need to exploit its **attributes** and **relations**. Link Mining help us to achieve this and as such is the next logical step in understanding how to find the underlying patterns in the RDF graph and how this will help us to infer knowledge not available at plain sight.

"Links," or more generically relationships, among data instances are ubiq-

uitous. These links often exhibit patterns that can indicate properties of the data instances such as the **importance, rank, or category** of the object. In some cases, not all links will be observed; therefore, we may be interested in predicting the **existence of links** between instances. In other domains, where the links are evolving over time, our goal may be to predict whether a link will exist in the future, given the previously observed links. By taking links into account, more complex patterns arise as well. This leads to other challenges focused on discovering substructures, such as **communities, groups, or common subgraphs**. Traditional data mining algorithms such as association rule mining, market basket analysis, and cluster analysis commonly attempt to find patterns in a dataset characterized by a collection of independent instances of a single relation. This is consistent with the classical statistical inference problem of trying to identify a model given a independent, identically distributed (IID) sample. One can think of this process as learning a model for the node attributes of a homogeneous graph while ignoring the links between the nodes. *A key emerging challenge for data mining is tackling the problem of mining richly structured, heterogeneous datasets*[16]. These kinds of datasets are best described as networks or graphs. The domains often consist of a variety of object types; the objects can be linked in a variety of ways. Thus, the graph may have different node and edge (or hyperedge) types. *Naively applying traditional statistical inference procedures, which assume that instances are independent, can lead to inappropriate conclusions about the data.* Care must be taken that potential correlations due to links are handled appropriately. In fact, *object linkage is knowledge that should be exploited.* This information can be used to improve the predictive accuracy of the learned models: attributes of linked objects are often correlated, and links are more likely to exist between objects that have some commonality. In addition, the graph structure itself may be an important element to include in the model. Structural properties such as degree and connectivity can be important indicators. Link mining is a recent research area that is at the intersection of the work in link analysis, hypertext and web mining, relational learning and inductive logic programming, and graph mining. [16] compile eight link mining tasks that can be broadly categorized as tasks that focus on objects, links, or graphs, these can be found in the following list 2.3.

1. Object-Related Tasks

(a) Link-Based Object Ranking

(b) Link-Based Object Classification

(c) Object Clustering (Group Detection)

(d) Object Identification (Entity Resolution)

2. Link-Related Tasks

   (a) Link Prediction

3. Graph-Related Tasks

   (a) Sub-graph Discovery

   (b) Graph Classification

   (c) Generative Models for Graphs

While data representation and feature selection are significant issues for traditional machine learning algorithms, data representation for linked data is even more complex.[15] In the following sections, we will assume that a data representation has been selected, that the designation of the objects or nodes in the graph has been made, and that the links or edges in the graph have been defined. However, when applying link mining to real world domains, one should not underestimate the effort required in choosing an appropriate representation.

Object-Related Tasks

Link-Based Object Ranking — Perhaps the most well known link mining task is that of link-based object ranking (LBR), which is a primary focus of the link analysis community. The objective of LBR is to exploit the link structure of a graph to order or prioritize the set of objects within the graph. Much of this research focuses on graphs with a single object type and a single link type. In the context of web information retrieval, the PageRank and HITS algorithms are the most notable approaches to LBR. *Link-Based Object Classification* — Traditional machine learning has focused on the classification of data consisting of identically structured objects that are typically assumed to be IID. Many real-world datasets, however, lack this homogeneity of structure. The discerning feature of LBC that makes it different from traditional classification is that in many cases, the labels of related objects tend to be correlated. The challenge is to design algorithms for collective

classification that exploit such correlations and jointly infer the categorical values associated with the objects in the graph. *Group Detection* — A third object-centric task is group detection. The goal of group detection is to cluster the nodes in the graph into groups that share common characteristics. A range of techniques have been presented in various communities to address this general problem. In recent years, a central challenge has been to develop scalable methods that can exploit increasingly complex graphs to aid the knowledge discovery process. *Object Identification* — The final object-centric task is entity resolution, which involves identifying the set of objects in a domain. The goal of entity resolution is to determine which references in the data refer to the same real-world entity. Examples of this problem arise in databases (deduplication, data integration), natural language processing (co-reference resolution, object consolidation), personal information management, and other fields. The problem has been defined with many variations; in the most general form, neither the domain entities nor the number of such entities is assumed to be known.

Link-Related Tasks

Link Prediction — We next turn to edge-related tasks. Link prediction is the problem of predicting the existence of a link between two entities, based on attributes of the objects and other observed links. Examples include predicting links among actors in social networks, such as predicting friendships.

Graph-Related Tasks

Sub-graph Discovery — An area of data mining that is related to link mining is the work on subgraph discovery. This work attempts to find interesting or commonly occurring subgraphs in a set of graphs. Discovery of these patterns may be the sole purpose of the systems, or the discovered patterns may be used for graph classification. *Graph Classification* — Unlike link-based object classification, which attempts to label nodes in a graph, graph classification is a supervised learning problem in which the goal is to categorize an entire graph as a positive or negative instance of a concept. This is one of the earliest tasks addressed within the context of applying machine learning and data mining techniques to graph data. Graph classification does not typically require collective inference, as is needed for classifying objects and edges, because the graphs are generally assumed to be independently generated. Three main approaches to graph classification have been explored. These are based on feature mining on graphs, inductive logic programming (ILP), and defining graph kernels. *Generative Models for*

Graphs — Generative models for a range of graph and dependency types have been studied extensively in the social network analysis community. For directed graphs with a single object and link type, there are several major classes of random graph distributions discussed in the literature: Bernoulli graph distributions, conditional uniform graph distributions, dyadic dependence distributions, and p* models.

To try to overcome the problems that arise using conventional approaches, [10] combines a logical approach together with a probabilistic one. At a minimum, a formal language for link mining must be (a) relational and (b) probabilistic. Link mining problems are clearly relational, since each link among objects can be viewed as a relation. *First-order logic is a powerful and flexible way to represent relational knowledge.* Important concepts such as transitivity (e.g., "My friend's friend is also my friend"), homophily (e.g., "Friends have similar smoking habits"), and symmetry (e.g., "Friendship is mutual") can be expressed as short formulas in first-order logic. It is also possible to represent much more complex, domain-specific rules, such as "Each graduate student coauthors at least one publication with his or her advisor." Most link mining problems have a great deal of uncertainty as well. *Link data is typically very noisy and incomplete.* Even with a perfect model, few questions can be answered with certainty due to limited evidence and inherently stochastic domains. The standard language for modeling uncertainty is probability. In particular, probabilistic graphical models have proven an effective tool in solving a wide variety of problems in data mining and machine learning. Since link mining problems are both relational and uncertain, they require methods that combine logic and probability. Neither one alone suffices: firstorder logic is too brittle, and does not handle uncertainty; standard graphical models assume data points are i.i.d. (independent and identically distributed), and do not handle the relational dependencies and variable-size networks present in link mining problems. **Markov logic** is a simple yet powerful generalization of probabilistic graphical models and first-order logic, making it ideally suited for link mining [10]. A Markov logic network is a set of weighted first-order formulas, viewed as templates for constructing Markov networks, therefore the authors propose to approach Link Mining problems using this type of model to *learn* and *infer* from the graph, finally in their work they exemplify the power of this technique, showing how can it be applied to classification problems.

In many current studies and applications, linked data are used to describe

systems consisting of interacted objects. Given that each node represents an object, in linked data, node attributes contain features, preferences, and actions, and links describe interactions between nodes. For instance, in a social network like Facebook, each user is viewed as a node. Node attributes may include gender, habits and number of friends (i.e. degree), and links can represent if two users are friends, if one comments on another's posts and other types of interactions. The great expressive power of linked data enables this data format to capture both characteristics and interactions of objects in various systems, such as linked gene mutation databases and ecommerce websites. Knowledge discovery on linked data is the process of leveraging both node attributes and link structures for the learning on the corresponding systems, and it is of great importance to understanding the characteristics and interaction patterns in these systems. One of the major challenges in mining linked data is how to effectively and efficiently utilize information from both node attributes and link structures. For this challenge, many existing models seek to represent link structures using selected statistics on networks, then combine selected statistics with node attributes. For example, in [23], [22], the authors characterized link structure using recursive egonet-based statistics, and further used the statistics to learn object roles. In [33], researchers detected spammers using degrees of friends. The primary limitation of these approaches is that the topological statistics in each task is usually subjectively selected. Therefore, such methods may miss critical patterns in linked data. Moreover, when the aimed tasks are complex, it could be very difficult to select or create relevant topological statistics. To avoid the limitation of the above methods, other approaches aim to extract a shared representation for both node attributes and link structures. For example, in [34], links are viewed as interactions between latent features of connected nodes. In [54], latent features are extracted with respect to the criterion that objects from different clusters are dissimilar while objects in the same clusters are similar. These methods usually rely on linear mappings to capture the relations among node attributes, network structures and the aimed latent feature representations of nodes. As a result, such approaches suffer from the simplicity of linear mappings and *fail to capture non-linear characteristics of nodes and links*. In order to address these issues, [31] propose a novel model named LRBM, which stands for Restricted Boltzmann Machines for Latent Feature Learning in Linked Data. Different from the aforementioned methods using graph statistics, the proposed model does not rely on any subjectively

selected topological statistics, and is capable of characterizing both node attributes and link structures in a unified framework. At the heart of this model is a shared latent feature representation of each node, which is used to formulate nonlinear relations among nodes, links and hidden units. To avoid large amount of sampling, Contrastive Divergence (CD) [24] is applied to train the model and other techniques such as fine-tune and parameter sharing are also used to simplify the calculation.

2.4 Graph Summarization

Graphs provide a powerful primitive for modeling real-world objects and the relationships between objects. Various modern applications have generated large amount of graph data. Some of these application domains are [55]:

- **Social networks** such as Facebook (www.facebook.com), Twitter (www.twitter.com), and LinkedIn (www.linkedin.com), these attract millions of users *(nodes)* connected by their friendships *(edges)*. Mining these social networks can provide valuable information on social relationships and user communities with common interests. Besides mining the friendship network, one can also mine the "implicit" interaction network formed by dynamic interactions (such as sending a message to a friend).

- **Market basket data**, such as those produced from Amazon (www.amazon.com) and Netflix (www.netflix.com), contain information about millions of products purchased by millions of customers, which forms a *bipartite graph* with edges connecting customers to products. Exploiting the graph structure of the market basket data can improve customer segmentation and targeted advertising.

- **Coauthorship networks and citation networks** constructed from DBLP ($www.informatik.uni-trier.de$) and CiteSeer (citeseer.ist.psu.edu) can help understand publication *patterns* of researchers.

- **The World Wide Web** can be naturally represented as a graph with nodes representing web pages and directed edges representing the hyperlinks. According to the estimate at www.worldwidewebsize.com, by May 15, 2009, the World Wide Web contains at least 30.05 billion

webpages. The graph structure of the World Wide Web has been extensively exploited to improve *search quality*, discover web *communities*, and *detect link spam*.

With the size of this networks, is difficult to grasp some of the underlying implicit and explicit structures. There is an overwhelming wealth of information encoded in these graphs, there is a critical need for tools to summarize large graph data sets into concise forms that can be easily understood. Graph summarization has attracted a lot of interest from a variety of research communities, including sociology, physics, and computer science. It is a very broad research area that covers many topics. Different ways of summarizing and understanding graphs have been invented across these different research communities. These different summarization approaches extract graph characteristics from different perspectives and are often complementary to each other. [58] summarize the graph, from the *ontological point of view*, doing it based on the implicit ontology and information in the RDFS tags. Sociologists and physicists mostly apply statistical methods to study graph characteristics. The summaries of graphs are statistical measures, such as degree distributions for investigating the scale-free property of graphs, hop-plots for studying the small world effect, and clustering coefficients for measuring the clumpiness of large graphs. In the database research community, methods for mining frequent subgraph patterns are used to understand the characteristics of large graphs. The summaries produced by these methods are sets of frequently occurring subgraphs (in the original graphs). Various graph clustering (or partitioning) algorithms are used to detect community structures (dense subgraphs) in large graphs. For these methods, the summaries that are produced are partitions of the original graphs. Graph compression and graph visualization are also related to the graph summarization problem, the latter will be discussed in section 2.5 of this chapter. However the techniques discussed in this chapter, focuses on a graph summarization method that produces small and informative summaries, which themselves are also graphs. We call them **summary graphs**. These summary graphs are much more compact in size and provide valuable insight into the characteristics of the original graphs. For example, in Figure 2.8, a graph with 7445 nodes and 19,971 edges is shown on the left. Understanding this fairly small graph by mere visual inspection of the raw graph structure is very challenging. An example summary graph for the original graph is shown on the right of Figure 2.8. In the summary graph, each node represents a set of nodes

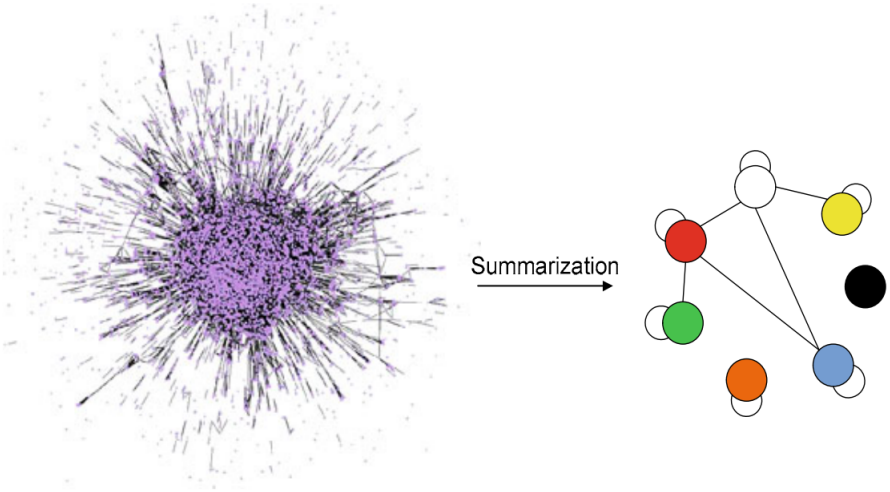


Figure 2.8: A summary graph (right) is generated for the original graph (left)

from the original graph, and each edge of the summary graph represents the connections between two corresponding sets of nodes.

The related problem of graph compression has been extensively studied, [55] presents two main methods for the compression problem **S-Node Representation of the Web Graph** The compression technique in employs a top-down approach to compute the S-Node representation, Figure 2.9. This algorithm starts from a set of supernodes that are generated based on the URL domain names, then iteratively splits an existing supernode by exploiting the URL patterns of the nodes inside this supernode and their links to other supernodes. However, different from the graph summarization method introduced in this chapter, this approach is specific to the web graph, thus are not directly applicable to other problem domains. **MDL Representation of Graphs** Essentially, this proposed representation is equivalent to the S-Node representation described above. The intranode graphs, positive superedge graphs, and negative superedge graphs in the S-Node representation, collectively, can produce the edge corrections needed to reconstruct the original graph from the summary graph. Based on Rissanen's minimum description length (MDL) principle, the authors formulated the graph compression problem into an optimization problem, which minimizes the sum of the size of the summary graph (the theory) and the size of the edge correction set (encoding of the original graph based on the theory). The representation with the minimum cost is called the MDL representation, Figure 2.10. Two

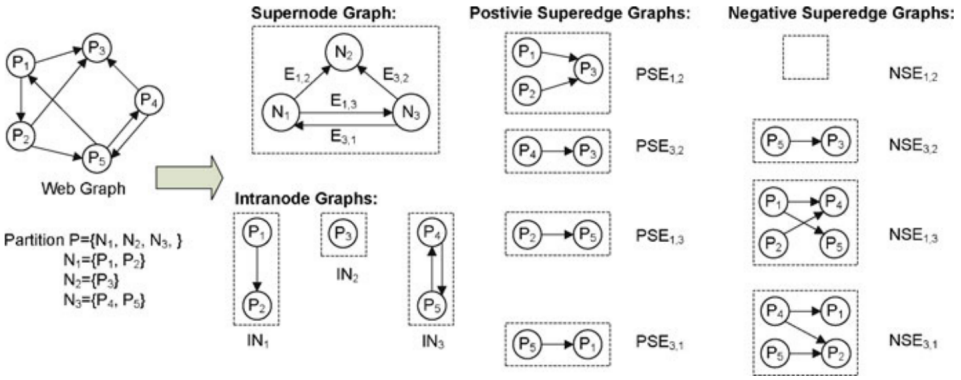


Figure 2.9: An example of S-Node representation (from [55])

heuristic-based algorithms are proposed in to compute the MDL representation of a graph. Both algorithms apply a bottom-up scheme: starting from the original graph and iteratively merging node pairs into supernodes until no further cost reduction can be achieved. The two algorithms differ in the policy of choosing which pair of nodes should merge in each iteration.

An RDF resource may have: no types, one or several types (which may or may not be related to each other). RDF Schema (RDFS) information may optionally be attached to an RDF graph, to enhance the description of its resources. Such statements also entail that in an RDF graph, *some data is implicit*. According to the W3C RDF and SPARQL specification, the semantics of an RDF graph comprises both its explicit and implicit data; in particular, SPARQL query answers must be computed reflecting both the explicit and implicit data. These features make RDF graphs complex, both structurally and conceptually. *It is intrinsically hard to get familiar with a new RDF dataset*, especially if an RDF schema is sparse or not available at all. Graph summarizations deals with: given an input RDF graph G, find an RDF graph S_G which summarizes G as accurately as possible, *while being possibly orders of magnitude smaller than the original graph.* Such a summary can be used in a variety of contexts: to **help an RDF application designer get acquainted with a new dataset**, as a first-level user interface, or as a **support for query optimization** as typically used in semi-structured graph data management, [58] approach is query-oriented so they work in a summary that enables static analysis and help formulating and optimizing queries; for instance, querying a summary of a graph should

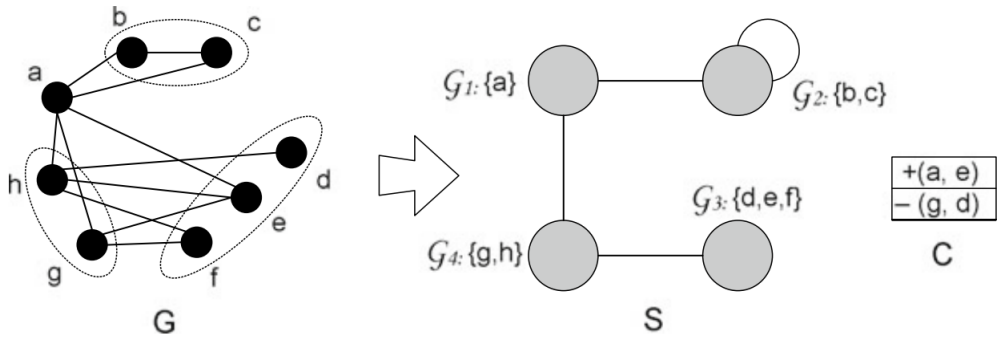


Figure 2.10: An example of MDL-based summary: G is the original graph, S is the summary graph, and C is the set of edge corrections (from [55])

reflect whether the query has some answers against this graph, or finding a simpler way to formulate the query etc; at the time of the article, it was the first semi-structured data summarization approach focused on partially explicit, partially implicit RDF graphs.

In [55] the authors present a very unique approach to an interactive querying scheme by allowing users to customize the summaries based on user-selected node attributes and relationships. Furthermore, this method empowers users to control the resolutions of the resulting summaries, in conjunction with an intuitive *"drill-down"* or *"roll-up"* paradigm to navigate through summaries with different resolution. This last aspect of drill-down or roll-up capability is inspired by the OLAP-style aggregation methods in the traditional database systems. Note that the method introduced in this chapter is applicable for both directed and undirected graphs, a modified version of the algorithm, will be the one used in this work and in later chapters the information regarding it, will be expanded.

2.5 Graph Visualization

As stated in previous sections, specially in Section 2.4, the amount of information contained in a graph can escalate very quickly and after a certain point it will be very difficult for the human eye to understand this amount of information [58] just by simply analyzing triples as tables without any other representations, this can prove difficult even when there is an *ontology* representing the data structure of the LD [58].

Despite the wealth of information contained in the Web of Linked Data, ordinary Web users, not familiar with Semantic Web technologies and the specific application domains, are difficult to directly consume this information resource [20]. Linked Data visualization can alleviate this problem [9], visualization can be a reasonable way to visually present the internal structure in the data and the relationship between the data; friendly visualization interfaces allow the users to identify any unreasonable, incorrect or duplicate data and links in the Linked Data [17], thus helping the users intuitively and efficiently analyze data. The authors in [9] summarized the general design guidelines and various user requirements for consuming Linked Data, and then introduced a set of usability criteria. Visualization tools that follow these usability criteria can help both tech-users and lay-users exploit Linked Data. After analyzing the existing Lined Data visualization methods and tools, they concluded that most existing methods/tools have been designed for tech-users and cannot provide an overview of the data, that is, they only meet certain aspects of the requirements and design guidelines. The work done in [11] introduced a prototype tool, Field, for using and visually analyzing large-scale Linked Open Data (LOD). The tool provides several visualization ways, but the user needs to manually specify complex instructions, hence it is difficult for lay-users to operate. Sgvizler [50] is a tool for rendering SPARQL query results in HTML pages, but users have to understand the source codes of the Web pages and the query statements; the tool's configuration process is somewhat complicated.

Linked Data Visualization Model (LDVM)

LDVM can be used to quickly make representations of RDF data visually [5]. It permits users to connect to and extract data from different datasets with different visualization techniques. The conceptual framework of the model is based on Data State Reference Model (DSRM) model offered by [8] as a result of adopting its stages' operations, names and transformations so as to fit the LD environment. The LDVM model may well be seen as a pipeline and it consists of four stages to process data, and three transformation operations in between these stages as depicted in Figure 2.11. The stages are RDF data: the raw data, Analytical extraction: to extract data from a previous stage, Visual abstraction: the visualizable data and View: present the data in different views. While the operations in between these stages are: Data

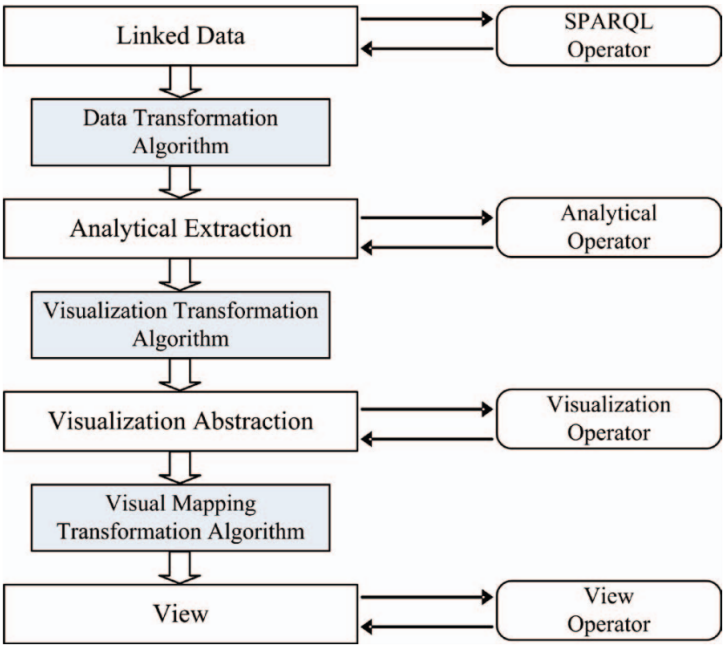


Figure 2.11: The LDVM model (from [38])

transformation, Visualization transformation, and Visual mapping transformation. So, the model can be sectioned into two core sections: data space and visual space. To proof the LDVM concept, a prototype and a useful RDF data browser is built based on the LDVM model called LODVisualization. It gives multiple visualizations while browsing RDF datasets. Any endpoint supporting JSON and SPARQL 1.1 is compatible with this prototype. Several visualizations, such as charts, tables, treemaps, can be shown. The server side of LODVisualization is written in Python, whereas the client-side built using HTML, CSS, and JavaScript mostly using D3.js library17 and InfoVis Toolkit18 . Google App Engine (GAE)

LODWheel is a LOD visualization tool, using two JavaScript libraries. But this tool does not support formal SPARQL query and detailed, deep data analysis, and it has a poor scalability. Currently, most LOD datasets provide their SPARQL endpoint query services, through which users can request useful RDF data. Therefore, using these query services in visualization tools is a key to consuming Linked Data [9]. Brunetti et al in [5] proposed a universal Linked Data Visualization Model (LDVM), which supports different dataset connections and visualization ways and is based on a visualization workflow, including analytical extraction and visual abstraction, to

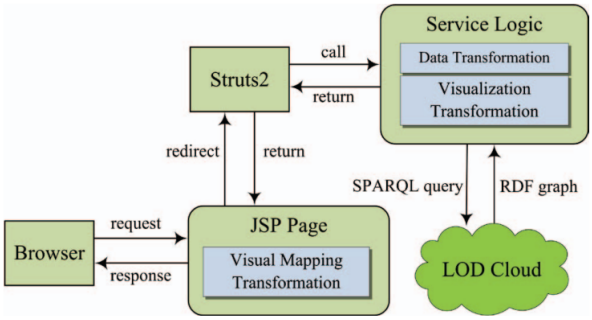


Figure 2.12: The LODViewer architecture (from [38])

achieve flexibility and automation. Ni et al in [38] based on the LDVM presented in [9] design Linked Data visualization algorithms, and develop a lightweight, easy-to-use prototype tool, *LODViewer*, using the platform independent JavaScript language. LODViewer can visualize different sources of RDF data including SPARQL endpoints for Linked Open Data (LOD) sources, and display the data in different graphic illustrations, in their work they verified the effectiveness and realizability of the proposed method. The time complexity analysis and experimental test show that the run-time of the proposed algorithms approximately exhibits a linear growth rate as the visualized RDF triples size increases. The general architecture used by their prototype system can be found in figure 2.12.

There are pieces of research that work on the issue about RDF visualization. They aimed to operate a complex network in any visualization canvas to be friendly for general users. We first reviewed some network visualization tools. Motif Simplification [12] considered some topologies of subgraphs, and replaced them with basic shapes such as diamonds, crescent, and tapered diamonds. It intended to give a big picture of a network rather than the detail of node-link. Gephi Open Viz Platform [32] is a powerful visualization tool that generated a well-shaped layout of network, allowed users to filter nodes and links, and had an option to set colors according to user preference. Both tools are suitable for general networks, but they are not designed for dealing with RDF data. One important issue of RDF data is a large number of inferred links creating a hairball-like graph, so the tools should consider this behavior in order to simplify a graph. RDF Gravity [14] provided an interactive view. Users could zoom a graph to view much more detail, and get details of nodes in the focus area using text overlay. Next, Fenfire [4] gave

an alternative view of RDF. It displayed the full details of the focused node and its immediate neighbors, but the other links were faded away according to the distance from the focus node. Both RDF Gravity and Fenfire offered well-organized displays, but they do not point out the issue of redundant data from inferred triples. Moreover, IsaViz [43] is an interactive RDF graph browser that used graph style sheets to draw a graph. It provided meaningful icons describing the type of each node such as foaf:Person, and grouped its metadata into a table in order to reduce highly interlinked data. It also allowed users to filter some nodes or properties to sparsify a dense graph, but this task required human effort to select some preferred URIs one by one.

Karwan et al [25], presents some of the most important current tools for data visualization, most of them are based on the LVDM principle and they are exploratory tools in order for the user to understand better the LD space. Here we will describe some of the most relevant ones for our research, a full comparison can be found in their work and in Figure 2.13, there is a table with a comparison between the tools, using a common set of parameters. **LodLive** is an LDB which uses standards of the LD to navigate RDF resources with the aim of spreading the fundamentals of LD in dynamic visual graphs through a user friendly interface [6]. Within the application, resources located in different endpoints can be linked so as to discover unexpected connections. Also, inverse relations can be navigated even for different endpoints. **CubeViz** is a faceted navigation browser and an extension of OntoWiki10 tool to visualize statistical data represented in RDF [47]. Usually, statistical data sets known as data cubes or basically cubes are distributed as spreadsheets or bidimensional matrices. **LODmilla** is a generic LDB to discover and edit LOD with the ability to combine the features of textual and graph based LOD browsers [35]. The web application provides viewing and searching LOD graphs as well as other browsing services. Smart View, or **SView**, is a system that allows users to navigate entity descriptions for LD perceptively . It uses Lenses, a group of organized features, to cluster and organizes entity descriptions so as to support users find related information easily.

Takeda et al in [7], introduces an approach to sparsify a graph using the combination of three main functions: graph simplification, triple ranking, and property selection. These functions are mostly initiated based on the interpretation of RDF data as knowledge units together with statistical analysis in order to deliver an easily-readable graph to users. By implement-

System Name	LodLive	CubeViz	LODVisualization	LODmilla	SView	LD Viewer	DBpedia Mobile Explorer	DBpedia Atlas
Release Date	2012	2012	2013	2014	2014	2014	2015	2015
Web Address	http://en.lodlive.it	http://cubeviz.aksw.org	http://lodvisualization.appspot.com	http://munkapad.sztaki.hu/lodmilla	http://ws.nju.edu.cn/sview	http://ldv.dbpedia.org	Mobile app	http://wafi.iit.cnr.it/lod/dbpedia/atlas
Dataset(s)	Multiple	Any dataset contains statistics	Multiple	Multiple	DBpedia	Multiple	DBpedia	DBpedia
Visualization Technique	Graph & infobox	Charts & scatter plots	Treemap, tree, charts	Graph & infobox	Tables via Lenses	Property table	Graph and text	Map-like visualization & infobox
Faceted Navigation	Yes, in graph	Yes	No	Yes	No	No	No	No
Query Model	Lookup	Dimension element selection	Manual selection	Lookup	Lookup	Lookup	Lookup	Lookup
Query suggestion	Yes	No	No	No	No	No	No	No
LD editing utilities	Yes, Ontology verification	Yes, add/edit/delete properties to data, create/edit/import knowledge bases	No	Yes, add/edit/delete nodes/properties	No	Yes, Annotate using DBpedia Spotlight	No	No
LD View Save	No	No	No	Yes	No	No	No	No
Breadcrumb	No	Yes, sessions	Yes, sessions	Yes	No	No	No	No
Live Endpoints	Yes, Multiple	Yes	Yes	Yes	Yes	Yes	Yes	Yes
API	No	No	No	No	No	No	No	No

Figure 2.13: The Linked Data Browser comparison table (from [25])

ing a prototype they show the feasibility and how this kind of visualizations helps users to understand better the data contained in the graph. The general functional diagram of the LDB presented by the authors can be found in Figure 2.14.

To conclude, there are many tools in the current landscape of the LOD visualization, some of them apply the LDVM presented in [5] like the one presented in [38] but in general there is still no common workground on how the data should be extracted and presented to the the user in order to aid the process to understand the LD graph. Some authors are using mode advance transformation tools to increase the amount of useful information presented to the user like the ones that worked in [7].

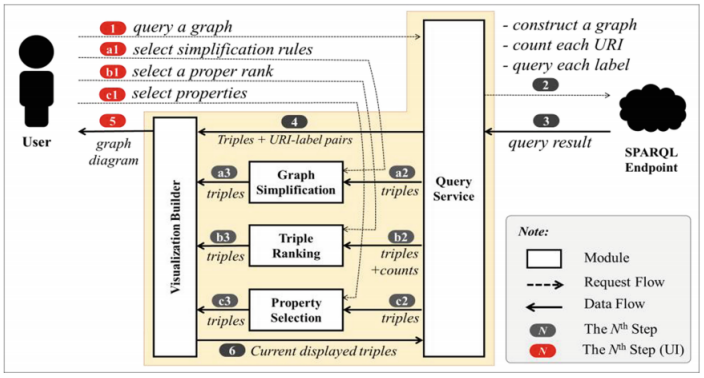


Figure 2.14: Functional diagram of the LOD viewer from [7]

Chapter 3

GSummarizer: Process model and algorithms

In this chapter, we will present the *GSummarizer* application logical model and the main algorithms that were used, the implementation details of the platform will be discussed in chapter 4. The model used for this application is based on the **LDVM** as seen in Figure 3.1. This starts by presenting the graph model used in the application, the second section gives an overview on how the LDVM is organized and the keypoints related to how was implemented in this work, then the explanation follows from top to bottom the model, starting by the *Extraction* algorithms used to obtained the data needed, in this case it is RDF data, then the *Data Transformation*, it is were the RDF data will be cleaned and transformed into a suitable representation for the next step, fifth, the *Data Visualization Transformation* algorithms, were the logical model will be transformed into a suitable model to be visualized by the user, in this section the main algorithms for *Graph Summarization* will be presented and how they were adapted in order to work better with the LOD and finally a brief overview how the *View* algorithms use the visualization model to present the information to the final user. Only the most important algorithms for this work will be presented in this Section with high technical detail.

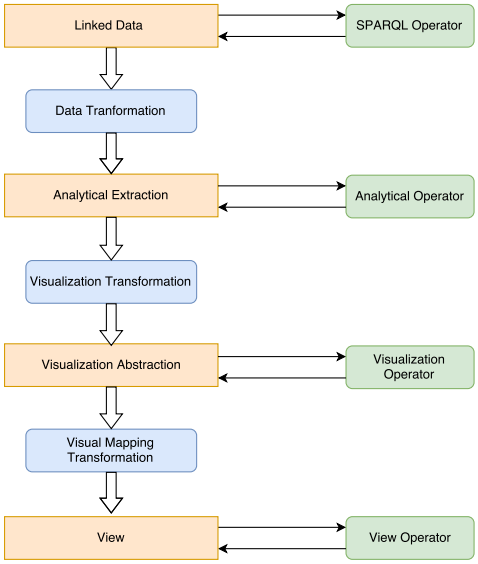


Figure 3.1: General Linked Data Visualization Model

3.1 The Graph model in the GSummarizer

The software solutions presented in this work, *GSummarizer*, is used to study the **SNAP** family of graph summarization algorithms in *Linked Open Data*, these were introduced in section 2.4 of chapter 2 and these particular family will be analyzed and presented in section 3.4 of this chapter. The algorithms from the *SNAP* family work with a simple kind of graph model given the input requirements of the algorithm (see Section 3.4), the graph model must have:

- Set of n nodes
- Each node can have 0 or n attributes.
- Each node can have 0 or n relations.

In general in a *Linked Data* graph, objects are represented by nodes, and relationships between objects are modeled as edges. In this work we follow this genera graph model guidelines, were the graph have objects (nodes) with associated attributes and different types of relationships (edges).

Definition 3.1.1 *General Graph*

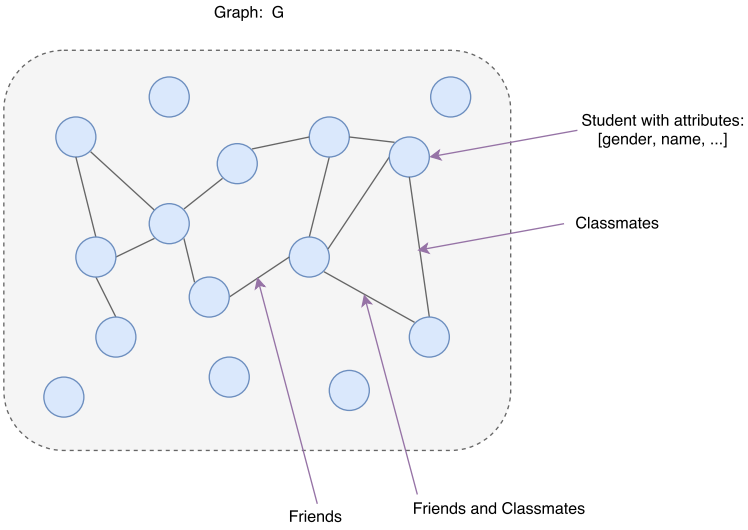


Figure 3.2: Graph model example

We denote a graph G as (V, Υ, Λ) where V is the set of nodes, and $\Upsilon = \{E_1, E_2, ..., E_r\}$ is the set of edge types, with each $E_i \subseteq V \times V$ representing the set of edges of a particular type. Nodes in a graph have a set of associated attributes, which is denoted as $\Lambda = \{a_1, a_2, ..., a_t\}$. Each node has a value for each attribute denoted as $v(a_i)$ where $v \in V$ and $a_i in \Lambda$. These attributes are used to describe the features of the objects that the nodes represent.

For example, in Figure 3.2, a node representing a student may have attributes that represent the student's gender and department. Different types of edges in a graph correspond to different types of relationships between nodes, such as friends and classmates relationships. Note that two nodes can be connected by different types of edges. For example, two students can be classmates and friends at the same time.

The model presented in Definition 3.1.1 is not specific enough regarding the structures that will hold the information of the node attributes and edges between nodes. In order to improve the model for this purpose we will introduce a more specific graph model. The following model was chosen in part, to accommodate to the needs of the *SNAP* algorithms, this will be further explain in Section 3.4.

Definition 3.1.2 *Attribute Matrix*

Given a graph $G = (V, \Upsilon, \Lambda)$ and the set of all possible attributes among all the nodes $v \in V$, we define the Attribute Matrix M_Λ *associated to G as*

$\{\lambda_{i,j}\}|_{i=0}^{|V|}|_{j=0}^{|\Lambda|}$ *where* $\lambda_{i,j} = a_j(i)$*, the value of the attribute* j *for the node* i*. Therefore, the cardinality of the* Attribute Matrix (AM) *is* $|V| \times |\Lambda|$.

Definition 3.1.3 *Relationship Matrix*

Given a graph $G = (V, \Upsilon, \Lambda)$*, the number of all the possible types of edges or relationship being* $|\Upsilon|$ *and* $k \in |\Upsilon|$*, we define the Relationship Matrix* M_Υ *associated to* G *as* $\{\upsilon_{i,j}\}|_{i=0}^{|V|}|_{j=0}^{|V|}$ *where* $\upsilon_{i,j} = E_k(i,j)$*, the Boolean value* $\{0,1\}$ *that determines if exist (1) or not (0) an edge or relation of type* k *between the nodes* i *and* j*. Therefore, the cardinality of the* Relationship Matrix (RM) *is* $|V| \times |V|$.

Definition 3.1.4 *Relationship Cube*

Given a graph $G = (V, \Upsilon, \Lambda)$*, the number of all the possible types of edges or relationship being* $|\Upsilon|$*, we define the Relationship Cube* C_Υ *associated to* G *as* $\{\upsilon_{i,j}^k\}|_{k=0}^{|\Upsilon|}|_{i=0}^{|V|}|_{j=0}^{|V|}$ *where* $\upsilon_{i,j}^k = E_k(i,j)$*. Therefore, the cardinality of the* Relationship Cube (RC) *is* $|\Upsilon| \times |V| \times |V|$.

Definition 3.1.5 *GSummarizer Graph*

Given a graph $G = (V, \Upsilon, \Lambda)$*, the associated Attribute Matrix and the associated Relationship Cube, we define the* GSummarized Graph Model (GG) G_{gs} *as* $(V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$.

In the GG model we define the structures that will hold the information of our graph, the definition above were based on the Associated Matrix of a graph and the theory found in [55] in order to better adapt the graph model to the algorithms that will use it in the next stage of the LDVM pipeline. From this moment on, we will refer as a Graph to any structure that complies with the GG definition.

3.2 The LDVM model in the GSummarizer

The logical model in the context of this work refers to the high level design of the process of the application, mainly, describing the stages or modules of the whole process, the flow of information and the responsibilities of each of the modules. Throughout this chapter it is assumed that the end-user of the system have had produced and input as a request that will be handled by an external module to the LDVM pipeline and therefore we only focus on the events after the user action.

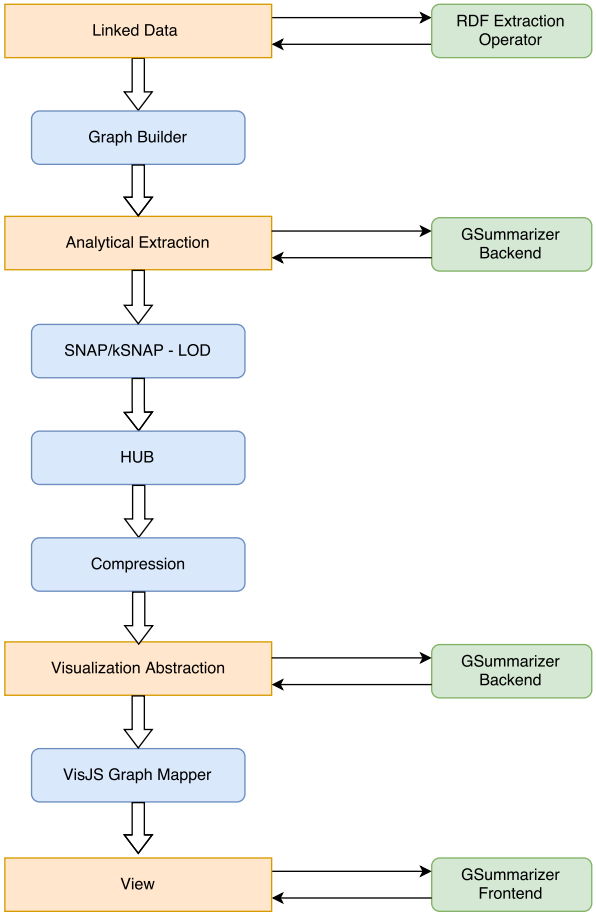


Figure 3.3: GSummarization current LVDM model

Even thought this work use the *SNAP* algorithms to analyze the behaviour of summarization in LOD, the GSummarizer support the extension of the transformation schemes and the type of data by making possible to use other extraction and transformation operators that could use a different type of graph model and summarization algorithms. This section will present a general overview of the algorithms used in each of the transformation operations, the stages of the model where described in the chapter 2 and will be followed as per the original design of the LDVM model.

1. **Data Transformation:** This transformation stage is composed of the RDF data extraction process and the Graph building process.

2. **Visualization Transformation:** In this stage the transformation algorithms must expect the output type of data from the previous stage in this case, is the graph model presented in Definition **??**. This stage is composed of the preprocessing algorithms used and adapted from the *SNAP* algorithms. The main transformation processes are focused on the *Summarization* of the graph: *SNAP/kSNAP* adapted to be Multi-attribute and Multi-relation, and adapted to the LOD problems. The final step of the transformation focus on the post-processing: first the HUB algorithm will further refine the grouping of the summarization clusters and finally the compression algorithm will clean the useless clusters.

3. **Visual Mapping Transformation:** This final transformation stage focus on mapping the output of the previous one into a suitable visualization scheme for the end user. This stage is highly technology dependent in this case, the processes and the technical details are out of the scope of this section, it will be briefly introduced at the end of this chapter and will be fully explained in chapter 4.

In this section we will present the overview of the process used to extract the raw data from the LOD endpoint. In the second part of the section, we will present the builder algorithm that builds the graph that will be used as the starting point of the *transformation process* that will be presented in Section 3.4. The modules involved in this process can be found in Figure 3.3

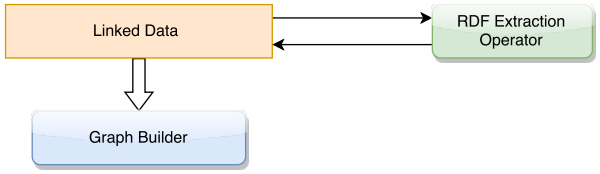


Figure 3.4: Data Transformation section of the GSummarization LVDM model

3.3 Data Transformation

In figure 3.4 the processes and stages related to the Data Transformation are depicted.

3.3.1 RDF Query Process

The GSummarization pipeline, in its current implementation, expects an **Extraction Operator** that will return a set of RDF triplets. In order to obtain these RDF triplets is necessary to *query* the data from an *RDF DB endpoint*, in this case , the extraction process is expecting a LOD endpoint as the purpose of this work is analyze the behaviour of LM algorithms in this type of data, for example *dbpedia.org*. The Extraction Operator must receive a **request** and will give a **response** accordingly.

The extraction request

LOD data as seen on section 2.1 is structured according to an *Ontology*. The *RDF query* will be based on the parameters that composed the Ontology, in this case: **Classes**, **Attributes** and **Relations**. An example of an Ontology structure is represented in the leftmost part of the figure 3.5, in this case a Person has attributes height, weight and relation type friend.

The RDF extraction method must receive a set of parameters $P = \{P_1, P_2, ..., P_n\}$ and a number l that refers to the number of instances per class that must be retrieved, in order query correctly the LOD end-point and ultimately build the graph correctly.

Definition 3.3.1 *LOD class*

We denote a LOD class C as (Λ, Γ), where $\Lambda_i = \{a_1, a_2, ..., a_m\}$ is the set of attributes types that belong to the class C and $\Gamma_i = \{r_1, r_2, ..., r_k\}$ is the set of relationship types that belong to C.

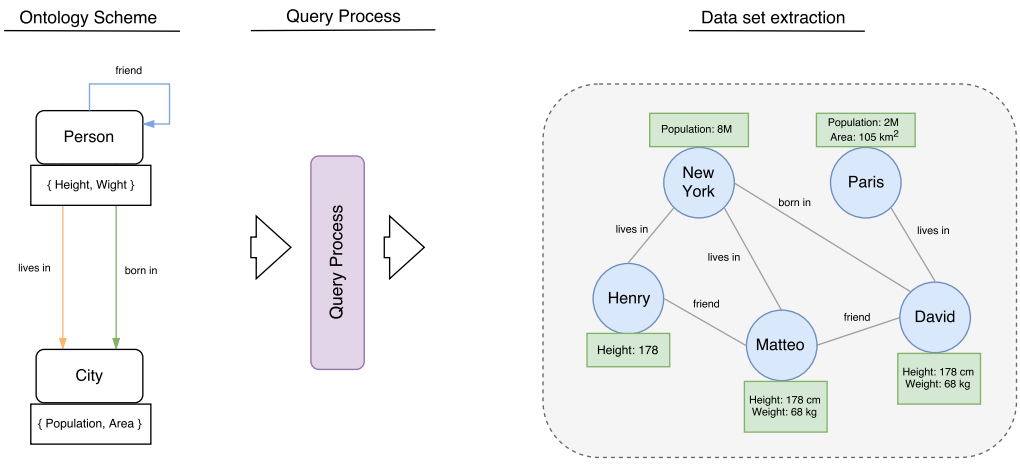


Figure 3.5: RDF Extraction logic

Definition 3.3.2 *RDF Request Parameters*

Be $C = \{C_1, C_2, ..., C_i, ..., C_r\}$ *the set of RDF classes in a LOD endpoint* EP_{LOD}. *We denote a request parameter* P_i *as* $(C_i, \lambda_i, \gamma_i)$ *where* $C_i = (\Lambda_i, \Gamma_i)$, $\lambda_i \subseteq \Lambda_i$ *and* $\gamma_i \subseteq \Gamma_i$.

The target classes

When you query the general LOD graph defined by the parameters P, you obtained will obtained a subgraph G with the nodes only contained in the subset of classes chosen classes C attributed Λ and Relation types Γ as depicted in Figure 3.6.

In order being able to extract more information using the same parameters we would need to expand the queried nodes, one strategy to do that is to make a one hop jump to **related classes**, in this case the related classes can be chosen based on the relationships Γ_i, this type of immediate related classes will be defined as one-hop relationships as shown in Figure 3.7. If we wanted to expand even more the class space we will need to make another jump but we don't have more information to do this due to the parameter information limit. Some of the options to overcome this are:

- Let the user to query two-hop relationships and further n-hop ones.
- Use some predictive algorithm in order to choose automatically which relations will be interesting for the user to explore based on the selection of relations.

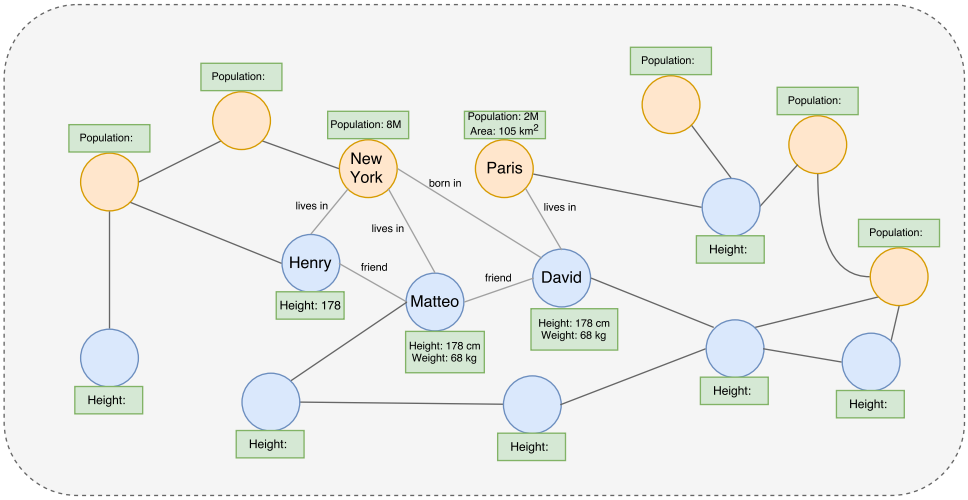


Figure 3.6: Queried subgraph

- Use some predictive algorithm in order to choose automatically which relations will be interesting for the user to explore based on the user profile, in this case it requires to get the profiling data before the query is made.

This options are out of the scope of this work and will be discussed in Section 6.1, left as future work. Therefore we define the expanded set of one-hop distance classes as the target classes to query nodes from. One of the issues with this approach is that the attributes and further relations of the expanded nodes will not be available at the time of the summarization, but this classes will be used in order to enhance the clustering at the time of the summarization process, this will be further explained in section 3.4.

The queries

The query process is performed by the *RDF Query Operator* module, the details are very technology dependent and will be expanded in Chapter 4; the specifics can be found in the reference implementation and will not be mentioned here given that they are out of the scope of this document. The general query set of operations are divided into two parts: the ones for the *Class selection* and the ones for the **Graph building**, the purpose of the first ones will be named briefly because the latter are the ones that are of most interest for this chapter:

The **Class selection** ones perform the following operations:

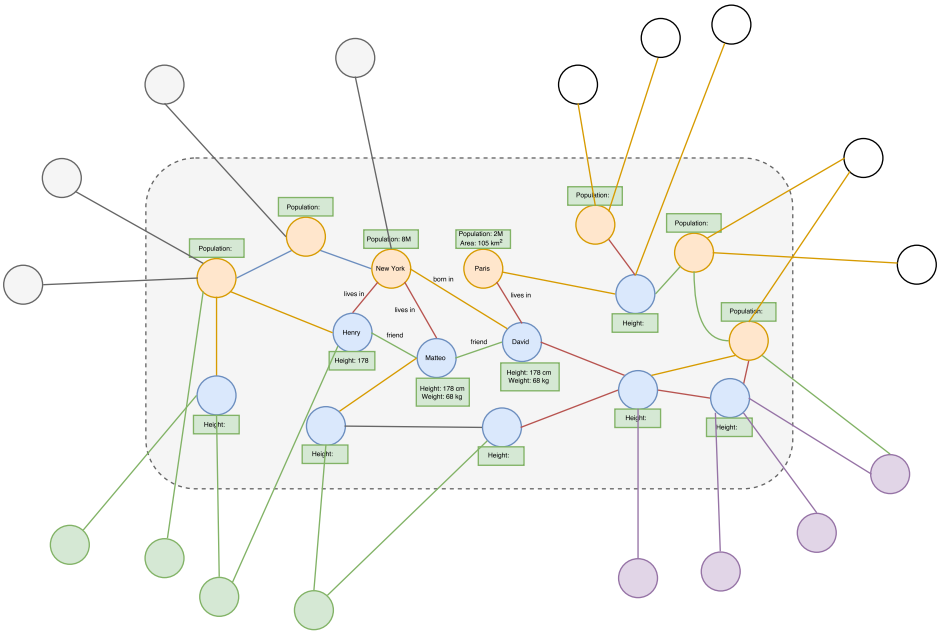


Figure 3.7: Queried subgraph with one-hop classes included

- Get all the available *classes* from the selected *LOD graph*
- Get all the available *attributes* and *relations* from the selected set of *classes*

The **Graph building** ones perform the following operations:

- **Instance from main classes** — Given a *class*, get n instances from that class. In the figure 3.6 the nodes represent the instances of the main classes, each one with the attributes and relationships.
- **Instance from related classes** — Given a set of *main class instances* and a *set of relationships of that instance*, get the connected nodes that are related to the main class instances through the set of relations. In the figure 3.7 the outer nodes will be the instances of the related classes, this nodes don't have relationships except for the ones connected with the main classes instances.
- **Attribute values for main class instances** — Given the set of a *main class instances* and a set of *the main class attributes*, get the values of the set of attributes for the main class for the set of instances.

In figure 3.6 we can see the value of the attributes for some of the classes, in this case, only the main classes, given that we don't have that information for the related classes (one-hop).

The specifics of some of the queries can be found in Section A.1.1. This queries will be contained in a module that will be described in Section **??**.

3.3.2 Graph Building

After understanding how to ask for the data needed for the *graph summarization* and *graph visualization process*, we will describe in this section, how the graph is build in accordance to the model presented in 3.1.5. In this section we will present in more detail the most relevant algorihtms for the GSummarization main process presented in this paper, the others can be found in the open repository of the project described in Section **??**

Algorithm 1: $BuildGraph(P, l, \epsilon)$

Input: P the set of parameters, l the number of instances to retrieve and ϵ the LOD endpoint.
Output: GSummarizer graph $G = (V, \Upsilon, \Lambda)$.
begin
  $V \longleftarrow \emptyset$
  for $p \in P$ **do**
    $CreateNode(V, p, l, \epsilon)$
  end
  $C_\Upsilon \longleftarrow \emptyset$
  $M_\Lambda \longleftarrow \emptyset$
  for $p \in P$ **do**
    $UpdateRelationshipCube(C_\Upsilon, V, p, l, \epsilon)$
    $UpdateAttributeMatrix(M_\Lambda, V, p, l, \epsilon)$
  end
  $G \longleftarrow Graph(V, C_\Upsilon, M_\Lambda)$
end

The *BuildGraph* algorithm first use the information contained in the set of parameters P to create the nodes $CreateNode$. For each parameter received a set of nodes is created. Not all the information regarding the parameters will be stored in each of the nodes, that is why the relationship and attributes have their own data structure, in this case the *relationship*

cube and the *attribute matrix*, in order to generate those there is a second pass throughtout the set of parameters updating the structures in each pass using $UpdateRelationshipCube$ and $UpdateAttributeMatrix$. Lastly using the updated structures, the *Graph* function puts it all together into the graph G

The $CreateNode$ function gets all the main classes and the related classes, with each one, it creates a new node.

Algorithm 2: $UpdateRelationshipCube(C_\Upsilon, V, p, l, \epsilon)$

Input: C_Υ the set relationship cube structure, V the list of nodes, p the request parameter, l the number of instances to retrieve and ϵ the LOD endpoint.

Output: GSummarizer graph $G = (V, \Upsilon, \Lambda)$.

```
begin
    for γ ∈ Γ_p do
        Δ ⟵ GetCouples(p, γ, l)
        for δ ∈ Δ do
            (i, j, f) ⟵ FindCouplesIndex(δ)
            if f == True then
                υ^γ_{i,j} = True
            end
        end
    end
end
```

The $UpdateRelationshipCube$ method, get the list of couples using $GetCouples$ function based on the current relationship γ and then per each couple, will find the indexes in the matrix using $FindCouplesIndex$. The latter function will stop as soon as it finds a couple, the worst case scenario will be searching all over the node space. The $UpdateAttributeMatrix$ is an even simpler algorithm, it cycles through the attributes in Λ_p getting the values for each of the attributes per each instance of a main class node.

3.4 Visualization Transformation

This part of Chapter 3 will present the logical specification for the base SNAP algorithm and the kSNAP algorithm, this means the algorithms as presented

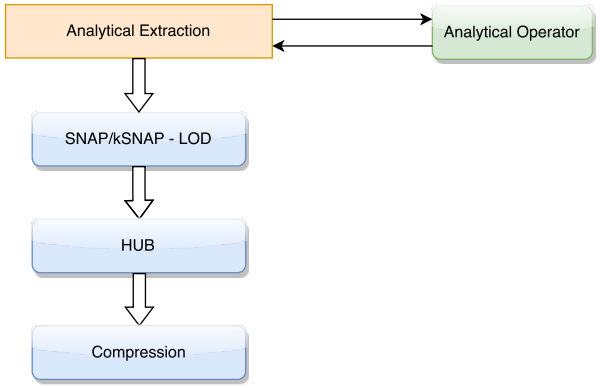


Figure 3.8: Visualization Transformation section of the GSummarization LVDM model

in their original work and their role in the achievement of the goals for this work, in this section the variable naming will follow the original work. The next section will present the main issues encountered while using this *summarization* method in the LOD and the modifications made to the original algorithm in order to overcome some of them. The following section will talk about the post processing methods use to refine the clustering *HUB algorithm* and to clean and compressed the obtained clusters given the characteristics of the data obtained at the end of the summarization process. In figure 3.8 the processes and stages related to the Data Transformation are depicted.

3.4.1 The SNAP algorithm

The authors in [55] define the set of nodes of graph G as $V(G)$, the set of attributes as $\Lambda(G)$, the actual value of attribute a_i for node v as $a_i(v)$, the set of edge types as $\Upsilon(G)$, and the set of edges of type E_i as $E_i(G)$. In addition, denoting the cardinality of a set S as $|S|$.

The SNAP operation produces a summary graph through homogeneous grouping of the input graphs nodes, based on user-selected node attributes and relationships. We now formally define this operation. To begin the formal definition of the SNAP operation, first the concept of node-grouping is defined as:

Definition 3.4.1 *Node-Grouping of a Graph*

For a graph G, $\Phi = \{G_1, G_2, ..., G_k\}$ *is called a* node-grouping *of* G, if and only if:

1. $\forall G_i \in \Phi, G_i \subseteq V(G)$ *and* $G_i \neq \emptyset$,

2. $\bigcup_{G_i \in \Phi} G_i = V(G)$,

3. $\forall G_i, G_j \in \Phi$ *and* $(i \neq j), G_i \cap G_j = \emptyset$.

Intuitively, a node-grouping partitions the nodes in a graph into non-overlapping subsets. Each subset G_i is called a group. When there is no ambiguity, we simply call a node grouping a grouping. For a given grouping Φ of G, the group that node v belongs to is denoted as $\Phi(v)$. It is further defined the size of a grouping as the number of groups it contains, also a partial order relation $\preceq$ is defined on the set of all groupings of a graph as:

Definition 3.4.2 *Dominance Relation*

For a graph G, the grouping Φ dominates the grouping Φ', denoted as $\Phi \preceq \Phi'$, if and only if $\forall G'_i \in', G_j \in \Phi$ s.t $G'_i \subseteq G_j$.

It is easy to see that the dominance relation $\preceq$ is reflexive, anti-symmetric and transitive, hence it is a partial order relation. Next a special kind of grouping is described, based on a set of user-selected attributes.

Definition 3.4.3 *Attributes Compatible Grouping*

For a set of attributes $A \subseteq \Lambda(G)$, a grouping Φ is compatible with attributes A or simply A-compatible*, if it satisfies the following: $\forall u, v \in V$, if $\Phi(u) = \Phi(v)$ then $\forall a_i \in A, a_i(u) = a_i(v)$*

If a grouping Φ is compatible with A, it is simply denote it as $\Phi(A)$. In each group of a A-compatible grouping, every node has exactly the same values for the set of attributes A. Note that there could be more than one grouping compatible with A. In fact a trivial grouping in which each node is a group is always compatible with any set of attributes. There is a global maximum grouping with respect to the dominance relation $\preceq$.This global maximum A-compatible grouping is denoted as Φ_A^{max}. Φ_A^{max} is also the A-compatible grouping with the minimum cardinality. In fact, if it is consider each node in a graph as a data record, then Φ_A^{max} is very much like the result of a group-by operation for these data records on the attributes A in the relational database systems.

The A-compatible groupings only account for the node attributes. However, nodes do not just have attributes, but also participate in pairwise relationships represented by the edges. Next, we consider relationships when

grouping nodes.For a grouping Φ, it is denoted the neighbor-groups of node v in E_i as $NeighborGroups_{\Phi,E_i}(v) = \{\Phi(u)|(u,v) \in E_i\}$. Now it is possible to define groupings compatible with both node attributes and relationships.

Definition 3.4.4 *Attributes and Relationships Compatible Grouping*

For a set of attributes $A \subseteq \Lambda(G)$ and a set of relationship types $R \subseteq \Gamma(G)$, a grouping Φ is compatible with attributes A and relationship types R or simply (A,R)-compatible, if it satisfies the following:

1. *Φ is A-compatible,*
2. *$\forall u, v \in V(G)$, if $\Phi(u) = \Phi(v)$, then $\forall E_i \in R$, $NeighborGroups_{\Phi,E_i}(u) = NeighborGroups_{\Phi,E_i}(u)$.*

If a grouping Φ is compatible with A and R, it is denote it as $\Phi_{(A,R)}$. In each group of an (A,R)-compatible grouping, all the nodes are homogeneous in terms of both attributes A and relationships in R. In other words, every node inside a group has exactly the same values for attributes A, and is adjacent to nodes in the same set of groups for all the relationships in R.

Given a grouping $\Phi_{(A,R)}$, we can infer relationships between groups from the relationships between nodes in R. For each edge type $E_i \in R$, it is defined the corresponding group relationships as $E_i(G, \Phi_{(A,R)}) = \{(G_i, G_j)|G_i, G_i \in , \Phi_{(A,R)}$ and $\exists u \in G_i, v \in G_j$ s.t. $(u,v) \in E_i\}$. In fact, by the definition of (A,R)-compatible groupings, if there is one node in a group adjacent to some node(s) in the other group, then every node in the first group is adjacent to some node(s) in the second. Similarly to attributes compatible groupings, there could be more than one grouping compatible with the given attributes and relationships. The grouping in which each node forms a group is always compatible with any given attributes and relationships. Among all the (A,R)-compatible groupings there is a global maximum grouping with respect to the dominance relation $\preceq$.

The global maximum (A,R)-compatible grouping is denoted as $\Phi^{max}_{(A,R)}$. $\Phi^{max}_{(A,R)}$ is also the (A,R)-compatible grouping with the minimum cardinality. Due to its compactness, this maximum grouping is more useful than other (A,R)- compatible groupings. Now we will review the **SNAP operation** as defined in its original work:

Definition 3.4.5 *SNAP Operation*

The SNAP operation takes as input a graph G, a set of attributes $A \subseteq \Lambda(G)$, and a set of edge types $R \subseteq \Upsilon(G)$, and produces a summary graph G_{snap}, where $V(G_{snap}) = \Phi^{max}_{(A,R)}$, and $\Upsilon(G_{snap}) = \{E_i(\ G, \Phi^{max}_{(A,R)}\ ) \mid E_i \in R\}$.

Intuitively, the SNAP operation produces a summary graph of the input graph based on user-selected attributes and relationships. The nodes of this summary graph correspond to the groups in the maximum (A, R)-compatible grouping. And the edges of this summary graph are the group relationships inferred from the node relationships in R. Before presenting the evaluation algorithm for the *SNAP operation*, it is necessary describe the fundamental data structures used in this and the other SNAP-like algorithms.

The SNAP data structures

Before presenting the *evaluation algorithm* it is necessary to define the data structures that are needed to perform the evaluation, these data structures can dynamically change during the snap process. The structures that will be declared here are commonly used in both *SNAP* and *kSNAP* algorithm.

Definition 3.4.6 *Neighbour-Group Bitmap*

Given a graph $G = (V, \Upsilon, \Lambda)$, a set of relations E and a grouping Φ, we define the Neighbour-Group Bitmap (NGB_Φ) associated to G and Φ as $\{ngb^k_{i,j}\}\ |^{|\Upsilon|}_{k=0}|^{|\Phi|}_{i=0}|^{|V|}_{j=0}$ where $G_i \subseteq \Phi$, $v_i \in V$, $k \in \Upsilon$ and

$$ngb^k_{i,j} = \begin{cases} 1 & if\ G_i \subseteq NeighborGroups_{\Phi,E_k}(v_i) \\ 0 & o.c \end{cases} \tag{3.1}$$

Therefore, the cardinality of the Neighbour-Group Bitmap (NGB) *is* $|\Phi| \times |\Upsilon| \times |V|$.

The NGB holds the information on the relationships between groups and nodes if such node has a relationship with the group. This structure can be updated fast because it holds only two possible values for the data. One of the objectives of the SNAP algorithm is to understand the links between groups and nodes, this can be done with the NGB, but in order to understand the relationships between nodes the SNAP method relies on the following the structure:

Definition 3.4.7 *Participation cube*

Given a graph $G = (V, \Upsilon, \Lambda)$*, a set of relations* E *and a grouping* Φ*, we define the Participation cube (* PC_Φ *) associated to* G *and* Φ *as* $\{pc^k_{i,j}\}|^{|\Upsilon|}_{k=0}|^{|\Phi|}_{i=0}|^{|\Phi|}_{j=0}$ *where* $G_i \subseteq \Phi$*,* $k \in \Upsilon$ *and*

$$pc^k_{i,j} = \sum_{\forall l} ngb^k_{i,l} \qquad l \in G_i \text{ and } G_j \in NeighborGroups_{\Phi,E_k}(l) \tag{3.2}$$

Therefore, the cardinality of the Participation cube (PC) *is* $|\Phi|\times|\Phi|\times|\Upsilon|$.

The PC quantifies the strength of the relationship between groups per each relationship k.

The SNAP evalutation algorithm

The evaluation algorithm that will be presented here is a modification from the original in order to handle the multiple relationship and attributes presented as options to the user. In the original work of [55] the technique and algorithm was defined for one relationship and one attribute but it signaled that the extension to multiple attributes and relationships should be simple. This topics will be addressed in section 3.4.3.

Algorithm 3: $SNAP(G)$

Input: G: a graph, where $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$
Output: Summarized graph.
begin
  $\Phi \longleftarrow MaxAttributeGrouping(G)$
  $NGB_\Phi \longleftarrow BuildNgb(G)$
  $PC_\Phi \longleftarrow BuildPc(G)$
  $UpdateDataStructures(NGB_\Phi, PC_\Phi, \Phi)$
  while $\exists\, v \in \{pc^k_{i,j}\}\ ,\ v \neq 0 \wedge v \neq |G_i|)$ **do**
    $\Phi \longleftarrow SnapSort(G, \Phi)$
    $UpdateDataStructures(NGB_\Phi, PC_\Phi, \Phi)$
  end
  $G_{snap} \longleftarrow BuildSummarizationGraph(G, \Phi)$
  return G_{snap}
end

The SNAP operation tries to find the maximum (A, R)- compatible grouping for a graph, a set of nodes attributes, and the set of relationships. The

evaluation algorithm starts calculating the maximum A-compatible grouping $MaxAttributeGrouping$, and iteratively splits groups in the current grouping, until the grouping is also compatible with the relationships. The algorithm for evaluating the SNAP operation is shown in Algorithm 3. In the first step, the algorithm groups the nodes based only on the attributes by a sorting on the attributes values. Then the data structures are initialized by this maximum A-compatible grouping using $UpdateDataStructures$. Note that if a grouping is compatible with the relationships, then all nodes inside a group should have the same set of neighbor-groups, which means that they have the same values in their rows of the bitmap. In addition, the participation array of each group should then only contain the values 0 or the size of the group $|G_i|$. This criterion has been used as the terminating condition to check whether the current grouping is compatible with the relationships. If there exists a group whose participation array contains values other than 0 or the size of this group, the nodes in this group are not homogeneous in terms of the relationships. We can split this group into subgroups, each of which contains nodes with the same set of neighbor-groups, this is achieved by $SnapSort$. After this division, new groups are introduced. One of them continues to use the same group id of the split group, and the remaining groups are added to the end of the group-array. Accordingly, each row of the bitmap has to be widened. The nodes of this split group are distributed among the new groups. As the group memberships of these nodes are changed, the bitmap entries for them and their neighbor nodes have to be updated using again the $UpdateDataStructures$ method. Then the algorithm goes to the next iteration. This process continues until the condition does not hold anymore. The nodes in the summary graph corresponds to the groups in the result grouping. Therefore the nodes will be group, starting from the grouping Φ_A^{max} until the $\Phi_{(A,R)}$ one.

3.4.2 The kSNAP algorithm

The SNAP operation produces a grouping in which nodes of each group are homogeneous with respect to user-selected attributes and relationships. Unfortunately, homogeneity is often too restrictive in practice, as most real life graph data is subject to noise and uncertainty; for example, some edges may be missing because of the failure in the detection process, and some edges may be spurious because of errors. *Applying the SNAP operation on*

noisy data, in our case, can result in a large number of small groups, and, in the worst case, each node may end up an individual group, we will see that with LOD data, that is the case. Such a large summary is not very useful in practice. A better alternative is to let users control the sizes of the results to get summaries with the resolutions that they can manage. Therefore, we introduce a second operation, called *k-SNAP*, which relaxes the homogeneity requirement for the relationships and allows users to control the sizes of the summaries.

The relaxation of the homogeneity requirement for the relationships is based on the following observation. For each pair of groups in the result of the SNAP operation, if there is a group relationship between the two, then every node in both groups participates in this group relationship. In other words, every node in one group relates to some node(s) in the other group. On the other hand, if there is no group relationship between two groups, then absolutely no relationship connects any nodes across the two groups. However, in reality, if most (not all) nodes in the two groups participate in the group relationship, it is often a good indication of a strong relationship between the two groups. Likewise, it is intuitive to mark two groups as being weakly related if only a tiny fraction of nodes are connected between these groups. Based on these observations, we relax the homogeneity requirement for the relationships by not requiring that every node participates in a group relationship. But we still maintain the homogeneity requirement for the attributes, i.e. all the groupings should be compatible with the given attributes. Users control how many groups are present in the summary by specifying the required number of groups, denoted as k. There are many different groupings of size k compatible with the attributes, thus we need to measure the qualities of the different groupings. We propose the $\Delta - measure$ to assess the quality of an A-compatible grouping by examining how different it is to a hypothetical (A, R)-compatible grouping. Now in order to formalize these concepts to reach a suitable metric, the following definition are needed:

Definition 3.4.8 *Groups participating in a group relationship*

It is defined the set of nodes in group G_i that participate in a group relationship (G_i, G_j) of type E_t as:

$$P_{E_t,G_j}(G_i) = \{u | u \in G_i \; and \; G_j \; s.t \; (u, v) \in E_t\} \tag{3.3}$$

Definition 3.4.9 *Participation ratio*

Given a group relationship (G_i, G_j) of type E_t, their participation ration is defined as:

$$p^t_{i,j} = \frac{|P_{E_t,G_j}(G_i)| + |P_{E_t,G_i}(G_j)|}{|G_i| + |G_j|} \tag{3.4}$$

For a group relationship, if its participation ratio is greater than 50%, we call it a strong group relationship, otherwise, we call it a weak group relationship. Note that in an (A, R)-compatible grouping, the participation ratios are either 0% or 100%.

Definition 3.4.10 $\Delta - measure$

Given a graph $G = (V, \Upsilon, \Lambda)$, a set of attributes A and a set of relationships R, the $\Delta - measure$ of $\Phi_A = \{G_1, G_2, ..., G_k\}$ is:

$$\Delta(\Phi_A) = \sum_{G_i, G_j \in \Phi_A} \sum_{E_t \in R} (\delta_{E_t,G_j)}(G_i) + \delta_{E_t,G_i)}(G_j)) \tag{3.5}$$

where,

$$\delta_{E_t,G_j)}(G_i) = \begin{cases} |P_{E_t,G_j}(G_i)| & if\ p^t_{i,j} \leq 0.5 \\ |G_i| - |P_{E_t,G_j}(G_i)| & otherwise \end{cases} \tag{3.6}$$

Intuitively, the $\Delta - measure$ counts the minimum number of differences in participation's of group relationships between the given A-compatible grouping and a hypothetical (A, R)-compatible grouping of the same size. The measure looks at each pairwise group relationship: If this group relationship is weak ($p^t_{i,j} \leq 0.5$), then it counts the participation differences between this weak relationship and a non-relationship ($p^t_{i,j} = 0$); on the other hand, if the group relationship is strong, it counts the differences between this strong relationship and a 100% participation-ratio group relationship. The δ function, defined in Equation 3.11, evaluates the part of the Δ value contributed by a group G_i with one of its neighbors G_j in a group relationship of type E_t.

Definition 3.4.11 ***k-SNAP operation***

The k-SNAP operation *takes as input a graph G, a set of attributes $A \subseteq \Lambda(G)$, a set of edge types $R \subseteq \Upsilon(G)$ and the desired number of groups k, and produces a summmary graph G_{k-snap}, where: $V(G_{k-snap}) = \Phi_A$, $s.t\ |\Phi_A| = k$ and $\Phi_A = argmin_{\Phi'_A}\{\Delta(\Phi'_A)\})$, and $\Upsilon(G_{k-snap}) = \{E_i(G, \Phi_A) | E_i \in R\}$*

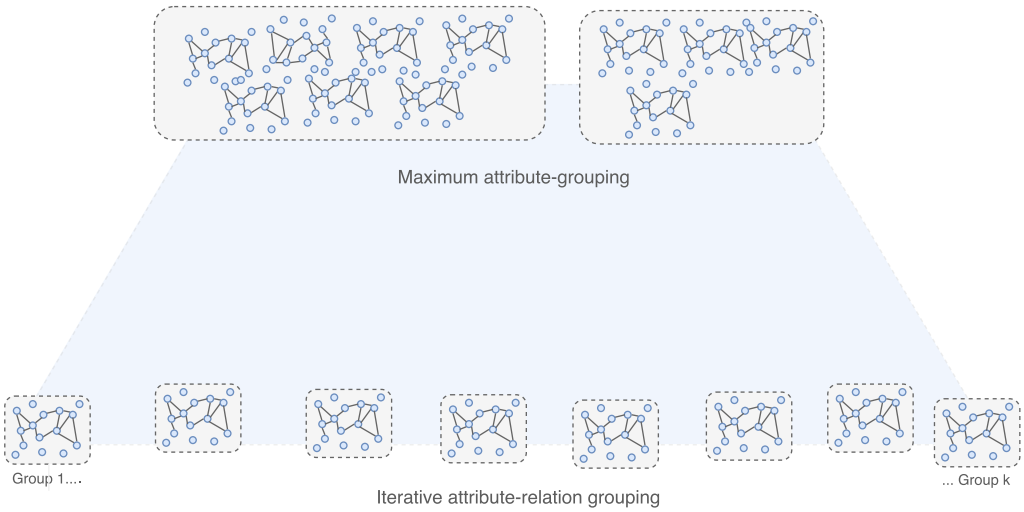


Figure 3.9: Scheme of the groups after a summarization ending with k elements

Given the desired number of groups k, the $k-SNAP$ operation produces an A-compatible grouping with the minimum Δ value.

The kSNAP evalutation algorithm

The $k-SNAP$ operation allows a user to choose k, the number of groups that are shown in the summary. For a given graph, a set of nodes attributes A and the set of relationship types R, a meaningful k value should fall in the range between $|\Phi_A^{max}|$ and $|\Phi_{A,R}^{max}|$. However, if the user input is beyond the meaningful range, i.e. $k < |\Phi_A^{max}|$ or $k > |\Phi_{A,R}^{max}|$, then the evaluation algorithms will return the summary corresponding to Φ_A^{max} or $\Phi_{A,R}^{max}$, respectively. For simplicity, it is assumed that the k values input to the algorithms are always meaningful.We can see an schema of how the nodes will be group starting from the Φ_A^{max} until the $\Phi_{(A,R)}^{k}$ in Figure 3.9.

In the work presented by [55] it is proven that computing the exact answer for the k-SNAP operation is NP-complete, therefore two incremental and heuristic algorithms were proposed to approximate to the solution, namely *top-down* and *bottom-up* approaches. The top-down approach starts from the maximum grouping only based on attributes, and iteratively splits groups until the number of groups reaches k. The other approach employs a bottom-up scheme. This method first computes the maximum grouping compatible with both attributes and relationships, and then iteratively merges groups until the result satisfies the user defined k value. In both approaches, we apply the

same principle: nodes of a same group in the maximum (A, R)-compatible grouping should always remain in a same group even in coarser summaries. We call this principle **KEAT** *(Keep the Equivalent Always Together)* principle. This principle guarantees that when $k = |\Phi^{max}_{A,R}|$, the result produced by the k-SNAP evaluation algorithms is the same as the result of the SNAP operation with the same inputs.

In the referenced work for the *k-Snap* evaluation algorithm, the approach that yield better results was the **Top-Down** one, therefore this is the one that we will present in this document and from now on any evaluation regarding the k-Snap algorithm will be based on such approach.

Algorithm 4: $kSNAP(G, k)$

Input: G: a graph, where $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$; k the required number of groups in the summary

Output: Summarized graph.

begin

$\Phi \longleftarrow MaxAttributeGrouping(G)$

$NGB_\Phi \longleftarrow BuildNgb(G)$

$PC_\Phi \longleftarrow BuildPc(G)$

$UpdateDataStructures(NGB_\Phi, PC_\Phi, \Phi)$

$SplitGroups(G, k, \Phi)$

$G_{snap} \longleftarrow BuildSummarizationGraph(G, \Phi)$

return G_{snap}

end

Similar to the SNAP evaluation algorithm, the top-down approach shown in Algorithm 4 also starts from the maximum grouping based only on attributes using $MaxAttributeGrouping$, and then iteratively splits existing groups until the number of groups reaches k with $SplitGroups$. However, in contrast to the SNAP evaluation algorithm, which randomly chooses a splittable group and splits it into subgroups based on its bitmap entries, the top-down approach has to make the following decisions at each iterative step: *(1) which group to split* and *(2) how to split it*. Such decisions are critical as once a group is split, the next step will operate on the new grouping. At each step, we can only make the decision based on the current grouping. We want each step to make the smallest move possible, to avoid going too far away from the right direction. Therefore,*we split one group into only two subgroups at each iterative step*. There are different ways to split one group into two. One natural way is to divide the group based on whether nodes

have relationships with nodes in a neighbor group. After the split, nodes in the two new groups either all or never participate in the group relationships with this neighbor group. This way of splitting groups also ensures that the resulting groups follow the *KEAT* principle.

Algorithm 5: $SplitGroups(G, k, \Phi)$

Input: G: a graph, where $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$; k the required number of groups in the summary; Φ the current grouping
Output: Split grouping Φ based on k
begin
  $H_\Phi \longleftarrow BuildHeap(G)$
  $UpdateDataStructures(NGB_\Phi, PC_\Phi, \Phi)$
  while $|\Phi| < k$ **do**
    $G_{max} \longleftarrow PopMaxCt(H_\Phi)$
    $G_{new} \longleftarrow Split(G, \Phi, G_{max})$
    $UpdateDataStructures(NGB_\Phi, PC_\Phi, \Phi)$
    $UpdateHeap(H_\Phi)$
  end
end

Now, the heuristic for deciding which group to split and how to split at each iterative step will be presented. As stated previously, the *k-SNAP operation tries to find the grouping with a minimum* $\Delta - measure$ *(see Equation 3.5) for a given k.* The computation of the $\Delta - measure$ can be broken down into each group with each of its neighbors (see the δ function in Equation 3.11). Therefore, the heuristic chooses the group that makes the most contribution to the Δ value with one of its neighbor groups. More formally, for each group G_i, we define $CT(Gi)$ as follows:

$$CT(G_i) = \max_{G_j}\{\delta_{E,G_j}(G_i)\} \tag{3.7}$$

Then, at each iterative step, we always choose the group with the maximum CT value to split and then split it based on whether nodes in this group G_i have relationships with nodes in its neighbor group G_t, where

$$G_t = \arg\max_{G_j}\{\delta_{E,G_j}(G_i)\} \tag{3.8}$$

As shown in Algorithm 5, to speed up the decision process, we build a **heap** on the CT values of groups using $BuildHeap(G)$. At each iteration, we

pop the group with the maximum CT value to split with $PopMaxCt(H_\Phi)$. Next $Split(G, G_{max})$ receives the group with the maximum CT value and the graph information, and splits G_{max} into two, based on the G_t obtained using Equation 3.8, returning an updated group referring to original G_{max} and a new group G_{new}. At the end of each iteration, we update the heap elements corresponding to the neighbors of the split group, and insert elements corresponding to the two new groups.

3.4.3 The modified kSNAP algorithm kSNAP-LOD

In this section we will present the modifications done to the **k-Snap** algorithm in order to optimize the summarization results obtained when used in a real world LOD data set. First we will analyze the current problems encountered in LOD while trying to summarize the generated graph. Next we will describe the challenges when using the specific *k-Snap* algorithm in LOD, and finally we will present the modifications made to the algorithm in order to improve the performance in some of the issues presented previously.

Incomplete data

In the tested LOD data sets, it is common to find instances of classes containing empty field attributes (ie. DBPedia (http://dbpedia.org/)). Attributes are at the core of the summarization process, relating similar classes to each other through the grouping Φ_A generated by $MaxAttributeGrouping$. The authors in [55] assume an *attribute array* without specifying the methods to tackle sparse data. This will cause that the *Attribute Matrix* M_Λ to have many empty values, therefore to have a lot of sparsity.

The problem of the sparsity of our data structures is that we are investing memory declaring them but the explicit information obtained is not useful in the plain form of empty spaces in our data structures.

Multiple attribute and relationships

In any real world exploration of a LOD DB, it is possible to query the graph on more than one class C_i at a time and more than one relation E_i and attribute a_i. In the present work by Tian et al [55], the algorithms don't specify how many attributes and relationships must be handled. In our case we have a set of classes where each one contains a set of attributes and

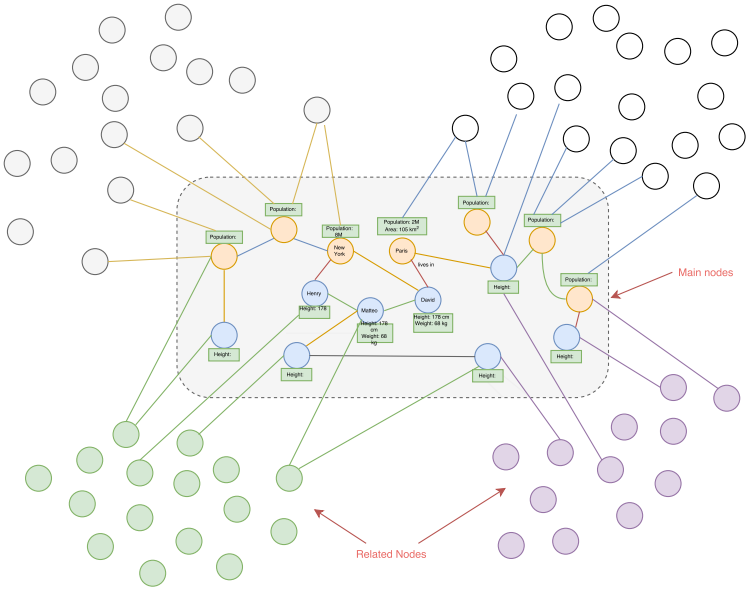


Figure 3.10: Scheme showing different classes instances either in the main-class nodes and in the related-class ones

parameters, therefore for each class C_i there will be a set of nodes V_i in the graph, therefore a set of indexes in the data structures holding the nodes indexes correspondingly, likewise for the **one hop** nodes.

- The **Relationship Cube** C_Υ, will hold all the indexes of all the nodes (main nodes or one hop nodes) involved in each relationship in Υ but in each M_Υ we will have as many empty spaces as nodes that are not involved in that relationship.
- The **Attribute Matrix** M_Λ, will hold all the indexes of all the nodes and the indexes of all the attributes in Λ, that means that there will be empty spaces for each node if it doesn't have all the attributes in Λ which it will be the most common case. This problem get worse when the problem of incomplete data is present.

Heterogeneous nodes

The problem when using only main-class instances is that given a relationship $\upsilon \in \Upsilon$ we will have $(u, v) \in E_\upsilon$ where $u \in C_i$, $v \in C_j$, $C_i \neq C_j$ and C_jC(our class set) (ie. C_i a main class and C_j is a related-class), but adding to the graph only C_i instances we will loose the information from the relation with

the C_j nodes, this is shown in Figure 3.10. In a *real world LOD dataset* the instances of a class are interlinked by relations and not self contained in an set of nodes of the same class, therefore contributing to the previously presented issues of sparsity of the *Relationship Cube* C_Υ.

Attribute similarity

As seen in the previous section, introducing multiple attributes will cause problems of sparsity and given the nature of real LOD data-sets, the incompleteness of data problem is also present. There is also another topic that is not specified in Tian et al [55], the attribute similarity measurement.

In the GSummarizer case we use a basic scheme to measure the similarity between two nodes based on their attributes, the main assumption here is that **all the attributes participating are numerical**, therefore the GSummarizer only handles comparisons between numerical attributes and only queries attributes that can be measured in this way. The similarity measure assumes that *the range of values of the attributes are unknown* and is defined as:

Definition 3.4.12 *Attribute similarity*

Given a graph $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$*, and two nodes* $v, w \in V$ *we define the attribute similarity between the two as:*

$$\theta(v, w) = \frac{\sum_{a_i \in \Lambda} dif(v, w, a_i)}{\sum_{a_i \in \Lambda} add(v, w, a_i)} \tag{3.9}$$

where,

$$dif(v, w, a_i) = \begin{cases} |a_i(v)| - |a_i(w)| & if\ |a_i(v)| + |a_i(w)| \neq 0 \\ 0 & otherwise \end{cases} \tag{3.10}$$

and

$$add(v, w, a_i) = \begin{cases} |a_i(v)| + |a_i(w)| & if\ |a_i(v)| + |a_i(w)| \neq 0 \\ 0 & otherwise \end{cases} \tag{3.11}$$

One-hop nodes

One of the objectives of this work is to give to the user a meaningful aid in order to understand a huge collection of data represented as a graph, such as a LOD data set. The k-Snap algorithm present a summary of a graph measuring the similarity between two nodes given its *relationships with other nodes* and its attributes. In this case two ways to enrich the information set for the summarization are:

1. *More nodes related to each other*, as we have seen previously, the lack of relationship information will affect the C_Υ and therefore the k-Snap algorithm will not be able to identify in a more accurate way to which group belongs each of the nodes, given that the participation ratio $p^t_{i,j}$ can vary greatly, therefore the $\Delta - measure$

2. *Complete attribute information*, as seen previously, a complete set of attribute values can make the Φ_A grouping more reliable due to the fact that it uses them to compare how similarly are two nodes attribute-wise.

As seen in Section 3.3.1, the query process involves *one hop* nodes because they enrich the relationship-based data structures, such as C_Υ. From the previously presented problems with the sparsity and incompleteness of attributes from the LOD data set, we face issues while making the grouping Φ_A, because all relations are not self-contained in the same class of nodes. This is one of the key issues that the k-Snap algorithm don't handle well in the reference work that we used. This will pose a problem given that the k-Snap and Snap relies on creating a coarse summary Φ^{max}_A and then refine it iteratively based on the relationship information contained in the relationship cube C_Υ. In order to obtain a more coarse initial clustering, more information is needed and therefore an strategy based on point (2) was chosen. The strategy was to query the one-hop nodes based on the selected relationships. This will increase the *class pool* and the size of the matrices but it will provide a richer set of relationships, all of this *without modifying the parameter information requested to the user*. In order to keep increasing the completeness of the node set we could have continued querying for the *two-hop nodes* but that would mean that (1) we should know the relationships that the related-nodes have and (2) let the user select again which ones the should be used in the second query process, both operations will make

the exploration of the graph more complex for the user and are out of the scope of this research. Another approach to a query *two-hop nodes* is, based on a pre-emptive analysis of the relationships of the *one-hop nodes*, select throught and automatic procedure the best relations to query from the *two-hop nodes*, this is an interesting approach that will be left for the Future work section in chapter 6. In the following section we will present the algorithm that will take advantage of this new set of information in order to perform a richer initial clustering.

3.4.4 HUB algorithm

After performing the attribute-based grouping operation and obtaining Φ_A^{max} we had attribute-based clusters of only main class nodes. This means that, initially we would have n set of main class-clusters but the same m related-class nodes as single node clusters as seen on Figure 3.11, this approach will not scale well, because on one side we need to reduce sparsity on the matrices and increase the information for the summarization process to be more effective. We device an algorithm, named *Hub algorithm* for this purpose based on the following characteristics:

- All the relationships are equal given that we don't have prior knowledge on them. So all relationships weight the same.
- The new clusters should be disjoint and can be composed of main-class nodes and related-class nodes

Now, lets introduce the concepts used in the algorithm:

Instead of giving an importance or wight to the relationships, given that we don't have any explicit information that can give us a sense of importance for them, we decide to search within the information that we had. The graph as any other highly heterogeneous network can have nodes with more connectivity than others and this is a central concept within the LOD networks, **connectivity**, this concept can be exploited to understand how two one-hop nodes or even main-class nodes are related to each other without having a direct connection through a relation. The central principles of the hub algorithm are the following:

- *Two nodes are related if both are connected to the same node (hub) and the more they are connected to the same node through different relationships, the more connected they will be with each other.*

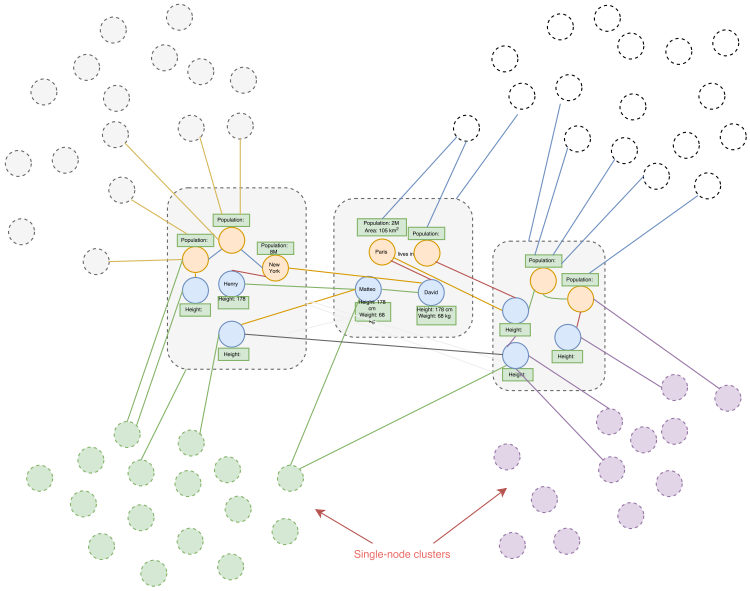


Figure 3.11: Scheme of the Φ_A^{max} grouping without using the hub algorithm, the related-classes clusters remain as single-node ones.

- *If two nodes are connected to the same hub , they are implicitly connected to the cluster that the hub node belongs to.*
- *A hub have will a higher value proportional to the number of connected nodes to it.*

Now lets ground these principles into definitions:

Definition 3.4.13 *Hub and Hub score*

Given a graph $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$, a hub can be any node $v \in V$ and the hub score $\mu(v)$ for $v \in V$ is defined as:

$$\mu(v) = \sum_{r \in \Upsilon} \sum_{w \in V, \forall w \neq v} \rho_r(v, w) \tag{3.12}$$

where,

$$\rho_r(v, w) = \begin{cases} 1 & if\ v_{v,w}^r = 1 \\ 0 & otherwise \end{cases} \tag{3.13}$$

Therefore $\mu(v)$ measures to how many nodes, v is connected, so we can associate each node to a hub score. With these score that we give to each

node, only the one above a threshold or non trivial ones will be taken into account. Now we will present the grouping algorithm that refines the clustering process of the Φ_A grouping.

Algorithm 6: $HubGrouping(G, \Phi)$

Input: G: a graph, where $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$; Φ the current grouping

Output: Refined grouping Φ_{hub}

begin
- $HH_\Phi \longleftarrow BuildHeap()$
- **forall** $v \in V$ **do**
  - $score_v \longleftarrow HubScore(v, C_\Upsilon, \Upsilon)$
  - $push(score_v, HH_\Phi)$
- **end**
- $v, score \longleftarrow pop(HH_\Phi)$
- **while** $score \neq 0$ *or* $\Phi = \emptyset$ **do**
  - $\phi' \longleftarrow CreateCluster(v, \Phi, C_\Upsilon, \Upsilon)$
  - **if** $\phi' \neq \emptyset$ **then**
    - $push(\phi', \Phi')$
  - **end**
  - $v, score \longleftarrow pop(HH_\Phi)$
- **end**
- $\Phi_{hub} \longleftarrow append(\Phi', \Phi)$

end

The $HubGrouping$ method first declares a Heap structure to hold the scores and retrieve them in an efficient way. The first $ForAll$ cycle will calculate the scores using the $HubScore$ method, this will perform the calculations based on the Equation 3.12. The next part of the algorithm focus on clustering the related nodes to the different hubs starting from the one that got the highest score, in this case, the algorithm will have a bias towards the highest scores given that the previously assigned nodes cannot be assigned again; to enforce this, the Φ' is used. In other words the strongest hub gets the nodes first. Each time the $CreateCluster$ is used, the nodes that were in previous clusters will be extracted from them in order to be part of the current hub cluster, if any of the clusters is empty, it will be removed from Φ. Also if the created cluster is empty, then it will not be added to the new set of clusters Φ'. Finally the new set of clusters will be added to the old ones in Φ_{hub}.

The modified k-Snap

The final modified k-Snap algorithm is defined as following:

Algorithm 7: $ModKSnap(G, k)$

Input: G: a graph, where $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$; k the required number of groups in the summary

Output: Summarized graph.

begin

$\Phi_A^{max} \longleftarrow MaxAttributeGrouping(G)$

$\Phi_{hub} \longleftarrow HubGrouping(G, \Phi_A^{max})$

$\Phi \longleftarrow \Phi_{hub}$

$NGB_\Phi \longleftarrow BuildNgb(G)$

$PC_\Phi \longleftarrow BuildPc(G)$

$UpdateDataStructures(NGB_\Phi, PC_\Phi, \Phi)$

$SplitGroups(G, k, \Phi)$

$G_{snap} \longleftarrow BuildSummarizationGraph(G, \Phi)$

return G_{snap}

end

The Algorithm 7 refines the clustering using the *HUB algorithm* and the modifications stated before in order to handle the sparsity and incompleteness problems.

3.4.5 The compression algorithm

After finishing the summarization, we have a graph $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$ with a grouping Φ in either of the cases mentioned before, the we use the method *BuildSummarizationGraph* in order to pack the information given by G and Φ and generate a $G_{summarization}$. This new graph can have all the information packed into a new structure:

Definition 3.4.14 *Raw summarized graph*

Given a graph $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$*, and a summarization grouping* Φ *we define as a raw summarize graph as:* $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda, \Phi)$

This summarized graph has all the information of the original graph with the following number of elements: $|V| + |V|^{2|\Upsilon|} + |V||\Lambda| + |\Lambda| + |\Upsilon| + |\Phi|$, bare in mind that each element holds different amount of data depending to which kind of data belongs to (ie. node).

This is a very heavy load for many systems and will not scale well specially for legacy devices and mobile phones that doesn't have a huge amount of memory and processing power at their disposal. In order to optimize for displaying performance we compress the information found in the original raw summarize node into a more compact summarization graph. The main idea of the algorithm is to use the groupings found in Φ as the new nodes of the graph and send only a summary of the data contained in each of them. The data summarization is highly dependant on the displaying technology used and is out of the scope of this document, an example of it can be found on the *experimental platform* used for the tests *GSummarization*. Therefore, the *BuildSummarizationGraph* will have to derivate methods, the first one defined as *BuildRawSummarizationGraph* based on Definition 3.4.14, the other one is *BuildCompressedSummarizationGraph* as seen in the Algorithm 8:

Definition 3.4.15 *Compressed summarized graph*

Given a graph $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$*, and a summarization grouping* Φ *we define as a compressed summarized graph as:* $G_{compressed} = (V', C'_\Upsilon, M'_\Lambda, \Upsilon, \Lambda)$ *where all the groups in* Φ *were transformed through a process* $\Theta(G_i) = v'_i$ *where* $G_i \in \Phi$*,* $v'_i \in V'$ *and the structures* C'_Υ *and* M'_Λ *by definition represents the relationship cube and the attribute matrix of the* $G_{compressed}$

This summarized graph only the information of the groups but not directly all the information of the each of the nodes from the original graph, therefore now the information contained in the graph has the following number of elements: $|\Phi| + |\Phi|^{2|\Upsilon|} + |\Phi||\Lambda| + |\Lambda| + |\Upsilon|$, in this case, depending on the effectiveness of the summarization we will require less space to store the graph, proportional to the size of Φ, if Φ is small, the compression will be more effective by following the Definition 3.4.15.

Algorithm 8: $BuildCompressedSummarizationGraph(G, \Phi)$

Input: G: a graph, where $G = (V, C_\Upsilon, M_\Lambda, \Upsilon, \Lambda)$; Φ the required number of groups in the summary

Output: Summarized graph.

begin

$V_{disconnected} \longleftarrow GetDisconnectedNodes(G)$

$G' \longleftarrow DeleteNodesFromGrouping(V_{disconnected}, G, \Phi)$

$G_{disconnected} \longleftarrow DeleteNodesFromGraph(V_{disconnected}, G')$

$\Phi_{disconnected} \longleftarrow UpdateClusters(G_{disconnected}, \Phi)$

$V_{new} \longleftarrow CreateNodesFromGroups(G_{disconnected}, \Phi_{disconnected})$

$G_{new} \longleftarrow AddNodesToGraph(G_{disconnected}, V_{new})$

$V_{delete} \longleftarrow GetAllNodes(\Phi_{disconnected})$

$G_{compressed} \longleftarrow DeleteNodesFromGraph(G_{new})$

return $G_{compressed}$

end

This algorithm is straightforward as a pipeline, starts by cleaning the graph from the nodes that are not connected to any other node. One of the key aspects while deleting nodes from the matrices or in general the data structures used in the algorithms presented here like the *relationship matrix* or the *attribute matrix* is to delete them from the biggest index to the lowest index. the second and third steps on the process focuses on deleting the nodes from the groups and then from the graph structures with $DeleteNodesFromGrouping$ and $DeleteNodesFromGraph$. Next, the algorithm creates nodes from the groups using a process Θ, in our case these process or method is defined in the reference implementation *GSummarizer*. Then, all the new nodes will be added to the graph $G_{disconnected}$ and finally all the nodes that were in the previous grouping Φ are deleted, finally we obtain a clean and compressed graph without any grouping extra information $G_{compressed}$

3.5 Visual Mapping Transformation

The visual mapping transformation (VMT) is highly dependant on the technology implemented in this case to visualize the graph and interact with the user. Conceptually this transformation is out of the scope of this document and the technical details can be found in the reference implementation of the

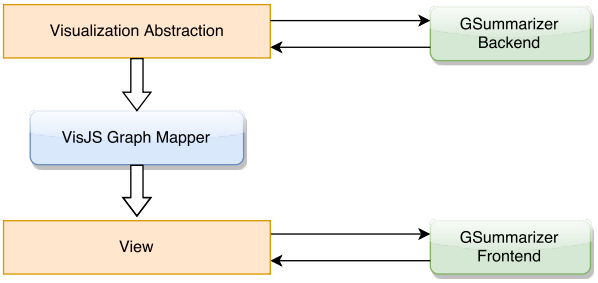


Figure 3.12: Visual Mapping Transformation section of the GSummarization LVDM model

experimental platform *GSummarizer*. In Figure 3.12 we can see the part of the LDVM that is involved in the VMT.

Chapter 4

Experimental platform: GSummarizer

In order to perform the test for our study, a testing platform was necessary, this chapter is devoted to the description of such testing application, it is called the *GSummarizer*. This chapter will start by presenting an overview of the platform, the motivations and the high level requirements of the platform. The next section will describe the general architecture of the *GSummarizer* , the technology stack, the design model and the motivation for each, after that, we will describe the front-end and back-end general overview. Finally the last two chapter will focus on the specifics of the front-end and back-end, describing the static and dynamic structure relevant to the work presented in this document.

4.1 GSummarizer Overview

4.1.1 Motivation

The main goal of this research is to understand the underlying problems of summarizing a LOD graph subset using LM techniques. Given the previous statement, we need to create a subset of the LOD graph, in other words, a sub-graph. A general scheme of the testing process is represented in Figure 4.1.

In our case, the subset must respond to the user input and as seen in Chapter 3, the user input was defined through a set of parameters P_i, these must be collected with the user input. The input can be obtained in many

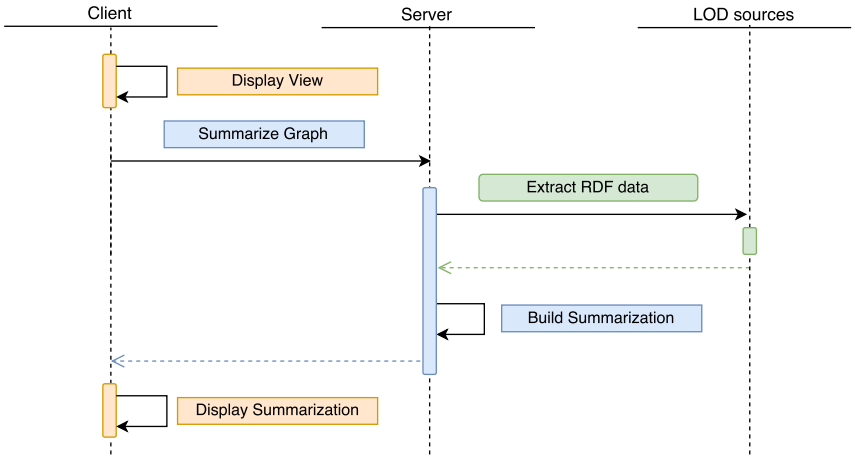


Figure 4.1: GSummarizer general process

ways, but in general this is handled by a user interface, therefore a module that manages the *user interactions* from the *user input*. The second part of the process is to build the target graph G_{target} to be summarized, based on the parameter set passed before by the user. The next part of the testing process is to *build the summarization* $G_{summarize}$ using the parameters P, following the *Separation of concerns (SoC)*, the summarization can be handled by a another module. Finally we need to display the results obtained in the summarization, this is another interaction that must be handled for the final user which is in-line with the responsibilities of the first module described before. In Section 3.2 we present the visualization model LDVM and how it will be applied to this work, so the modules involved in the platform must be responsible of enforcing the visualization model as well. From the previous analysis, the following are the requirements for the testing platform:

A Handling the interaction with the user: *user input* and *displaying the summarized graph*.

B Getting the LOD data needed based on the user input parameters.

C Performing the graph summarization process according to the LDVM model.

Based on the goals of this research and the future work in mind, the testing platform must be extensible in the following aspects:

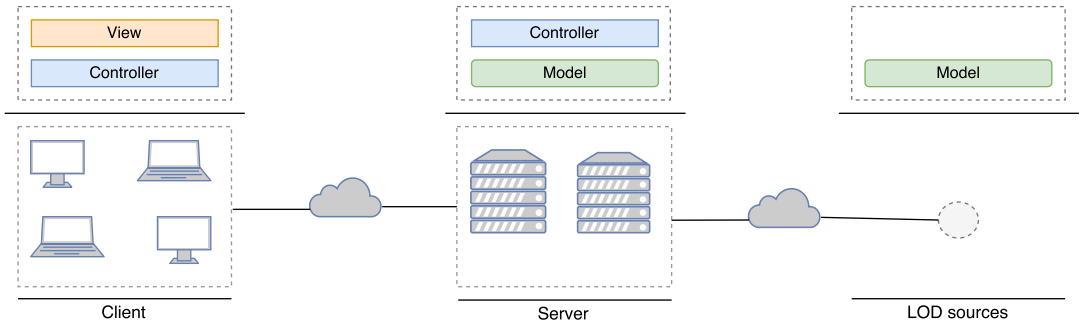


Figure 4.2: GSummarizer general architecture model

D Different user interactions, therefore having the possibility of extending the way that the user interfaces with the testing platform from the user graphical interface and input parameters.

E Being able to obtain data from different LOD sources.

F Process more than one type of LM algorithm.

G Displaying the graph in more than one way

There are different examples in Section 2.5 of visualization platforms compliant with the LDVM, but to *the best of our knowledge there is no single platform or application that we have access to, that can fulfill the following requirements*. Therefore we had to create one using the best technologies possible for our given requirements and within the constrains of this research.

4.1.2 Architectural Model

In Section 4.1.1 we talk about the principle of *Separation of concerns (SoC)* and we describe how the testing process should have different modules in charge of the specific requirements, therefore each module must handle different responsibilities, an initial separation of responsibilities in the process can be found in Figure 4.1. Data analysis can be very expensive computer-wise, this means that the computational power needed to perform the process (summarization of a graph in our case), when escalating the amount of data, will proportionally increase, (ie. exponentially or linearly) depending on the efficiency of the algorithms, this means that it may not be feasible to process the data in the same location where the user is. In the case of the *GSummarizer* it is reasonable to assume that the modules of the application that

handle the user interaction and the ones that handle the graph summarization, may not be in the same location (ie. cpu, server, cluster, ...etc), in order to account for this we choose to use a **Client-Server** architecture. Finally, following the *SoC*, and from Figure 4.1, three big components that handle different parts of the process: *Presentation/View*, *Summarization/-Logic* and *Data/Model* can be identified. Taking into account the previous components we can benefit from using an architecture following the **Model - View - Controller (MVC)** pattern. Therefore the *GSummarizer* will have a *Client-Server* architecture combined with an MVC-like architectural pattern as shown in Figure 4.2.

4.1.3 Technology Stack

Based on the architectural model defined for the *GSummarizer testing platform* the following technologies where chosen for the solution:

- Server-side programming language: **Python**, this language is one of the most used languages in data analysis, data mining and machine learning.
- Server-side application server: **Gunicorn**, this is a light WSGI HTTP web server build in python.
- Web application framework: **Django**, this python based framework follows the *MCV* and can be implemented with a *client-server* architecture in mind.
- Client-side languages and technologies: For the presentation layer of the application that it will be **web-based**, the languages will be *JavaScript*, *Html* and *CSS*
- Client-side framework: *Angular*, this javascript based framework follows the *MVC* pattern, is scalable and well maintained by *Google Inc.*
- Graph visualizer: *VisJs*, a JavaScript base graphic library that can handles graph visualization very efficiently.

This technologies were chosen thinking about the architectural requirements and the duration of the platform for testing *future developments* in this area.

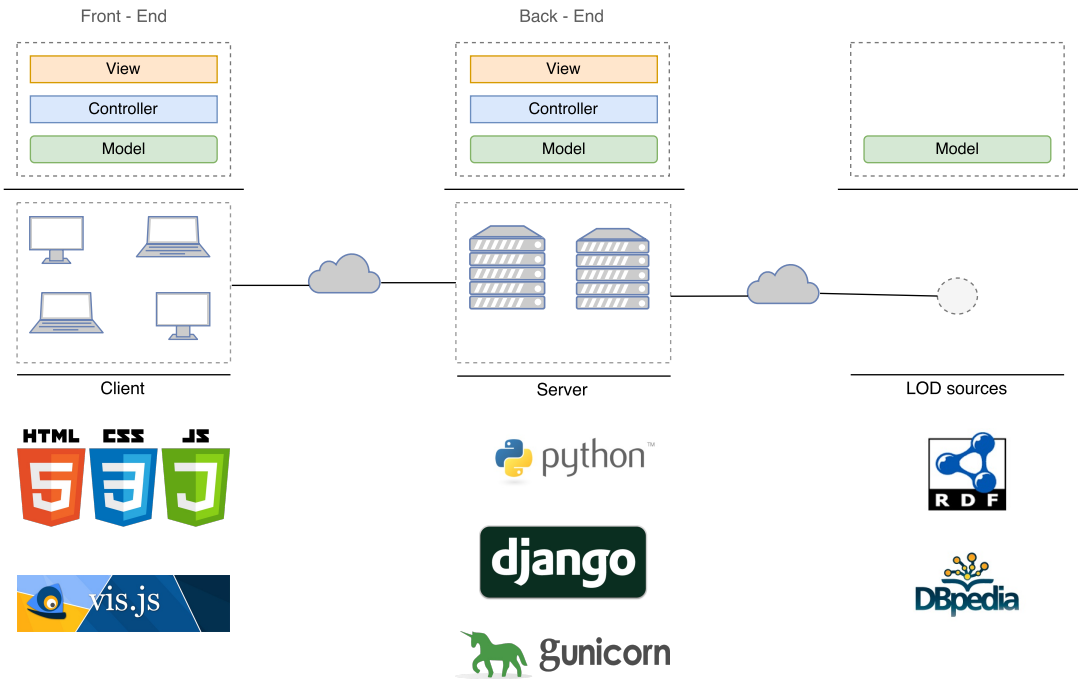


Figure 4.3: GSummarizer general architecture with technologies

4.2 General Architecture

In this section we will present the general architecture and design of the *GSummarizer* application, starting with the high level architectural view and going down until reaching the main modules that compose it. In each level the static and dynamic design decisions will be presented motivated. The description of each of the main modules will be left for Sections 4.3 and 4.4.

In Figure 4.3 we present the architecture of the application, specifying the MVC-like model used and the technologies involved in each layer of the platform. As stated before , the general architecture follows a *Client-server*, in the case of the *GSummarizer*, the *MVC-like* pattern is used in a distributed way. The server and the client have components belonging to the view, the controllers and the model, these components are shared but are disjoint components and have different responsibilities regarding the views, controllers and models, this means that the view component in the server have different and disjoint responsibilities than the ones in the client. The MVC distribution of responsibilities further allows the computation to be

balanced between the client and the server. The part of the model assigned to the LOD sources is a static one, but is of key importance to the testing platform overall, therefore it was included thought we cannot modify it directly.

4.2.1 The client

The client in the case of the *GSummarizer* is responsible of displaying the *View* that will give the information to the user to create a request, meaning, the set of parameters P, this view needs updating according to the user actions, therefore a *View Controller* is necessary; finally, the model is represented by the summarized graph, that will be updated according to the user actions in the view. To support the display of the summarized graph in the *view* the graphic library **VisJS** will be used. Therefore, the *controller* is in charge of handling the *users interaction*, sending the *request* to the server and displaying the *response* of it. The previously described scenario fulfills the requirement A from Section 4.1.1. The module that will handle the client side of the architecture will be named from now on as **front-end**. The front-end use the framework *AngularJs* , this framework use the MVC pattern. The explanation of the front-end will be expanded on Section 4.3.

4.2.2 The server

The *GSummarizer* server side follows the *MVC* architecture, as well as the client. In this case, the *Views* are represented by the exposed services, specifically the **REST** services that will be in charge of handling the clients requests. The main requests is the information necessary so the user can create a request, and , as seen on Figure 4.1, the summarization request. The *Controller* module in this case will handle the data extraction and the summarization logic. The *Model* is represented by the extracted data from the **LOD sources**. The model in this case is represented using RDF format and each LOD source will manage this format using their own *Stack* that is why a further description of the sources is not relevant for the current document. The *GSummarizer* was thought to handle different formats other than RDF to build the source if the correct *extraction* were to be implemented. The server-side component, from now on will be named as the Back-end of the *GSummarizer* and the specifics of it will be presented on Section 4.4

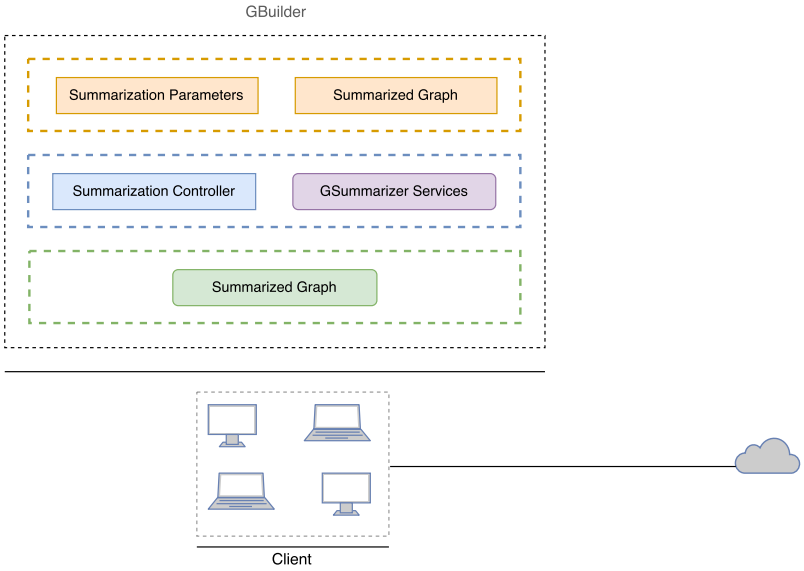


Figure 4.4: Front-end, GBuilder modules and its components

4.3 Front-End

The front-end responsibilities were stated in the previous section. The front-end use the services provided by the back-end in order to present a summarization to it. We use the *modified snap and k-snap* that require specific inputs, this may not be the case for all the summarization algorithms that can be tested in the platform, with that in mind the front-end can be extended with other modules that will include their own *View*, *Controller* and *Model* fulfilling the following requirements for the client-side module: [D], [F] and [G], presented in Section 4.1.1. The module in our case is the **GBuilder**. In figure 4.4 there is an schema of the main components of it.

The *GBuilder* module is composed of two views: *Summarization Parameters* and *Summarized Graph*, the first view interacts with the user by creating and sending the request parameters P for the summarization. The parameters will be chosen from the set of *available classes* with its *attributes and relationships*, these depending on the chosen *LOD dataset*. The second view is in charge of displaying the summarized graph. The controller layer of the *GBuilder* is composed of the *Summarization Controller* and the *GSummarization Services*, the first one is in charge of managing the update of the views due to the user interaction, either controlling how the views are displayed internally or using the services to provide the views with the data

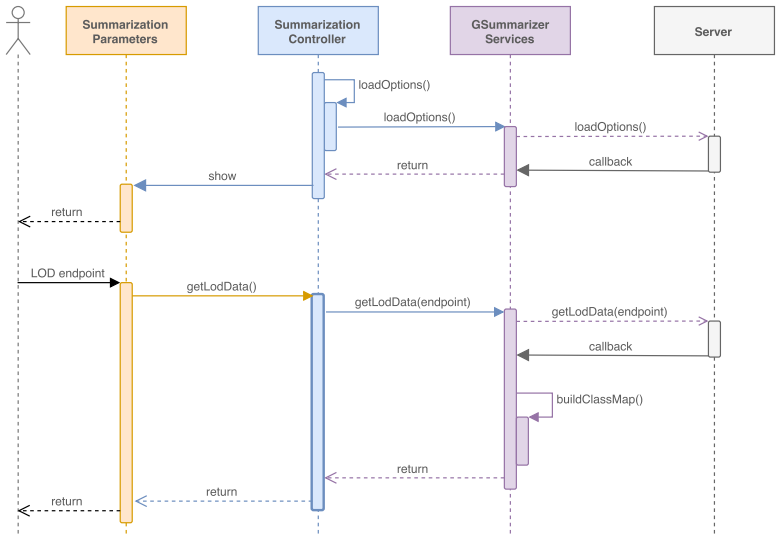


Figure 4.5: GBuilder sequence diagram for getting the parameter selection interface ready for the user after the LOD endpoint selection.

needed for the user. The service component is in charge of sending the request to the back-end and retrieving the answer possibly updating the stored model. In our case, the *model* that the front-end handles is represented by the summarized graph returned from the back-end.

Before any interaction with the user, by default , the web client will perform a request to the *application server (AS)*. The AS will provide an initial view while loading all the data needed for the view, in our case loading to the client all the modules required for the front-end of the *GSummarizer*. The scenario described above is the initial entry for the testing platform and all the sequence diagrams assume the starting point, the time after the initial request. In Figure 4.5 we can see the user waiting for the *Summarization Parameters* view to be ready, in order for that to happen, the controller must load all the information needed first (ie. list of all the LOD endpoints available). The next step after the view is loaded, is for the user to choose a LOD endpoint so all the classes, relationships and attributes can be loaded so the user can build the request parameters. The description of the structure of the parameters can be found in Section 3.3.1.

In Figure 4.6, the user has already selected the request parameters and send the signal to the view to submit the request for the summarization to be executed. The controller receive the signal and build the correct request based on the parameters selected by the user. To send the request the con-

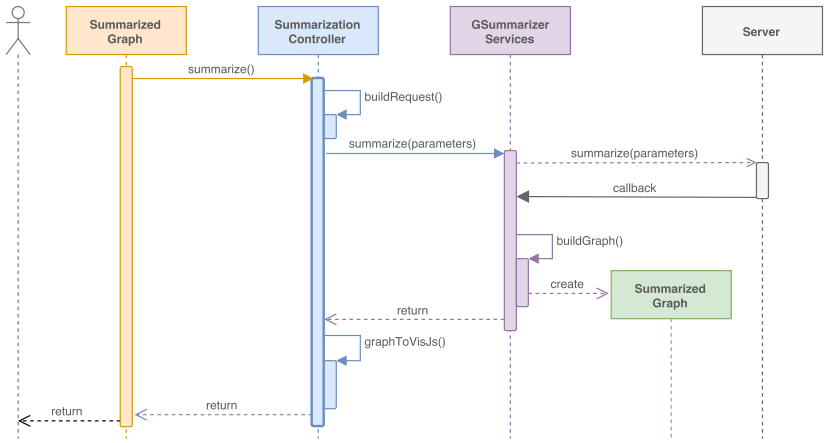


Figure 4.6: GBuilder sequence diagram for the events after submiting the request for the graph summarization.

troller use the *GSummarizer service*. The front-end will wait for the callback of the server, when this happens, the service will take the returned message and deserialize it into an object suitable for the controller to handle. The controller will take the object and it will process it ina a way that the visualization artifact (in this case VisJs) will be able to display it to the user in the *Summarized Graph* view.

4.4 Back-end

The GSummarizer back-end as stated on Section 4.1.3, uses DJango as the web application framework, this partially restricts the architecture to the framework design decisions, in our case we use DJango modules to orchestrate the use of the GSummarizer core modules and the the communication with the client. The *GSummarizer core modules* are the set of functional units that are responsible of fulfilling the requirements stated on Section 4.1.1. A high level detail of the composing modules of the back-end, can be seen in Figure 4.7 Each of the modules responsibilities and how they map to the previously defined requirements are presented in the following list:

- **Extraction** This module is responsible of getting the *required data* for the caller module. The general input of will be the *request parameters and the data source*. As output, the methods return a *data structure contained in the messages module*, this will enforce the *interface*

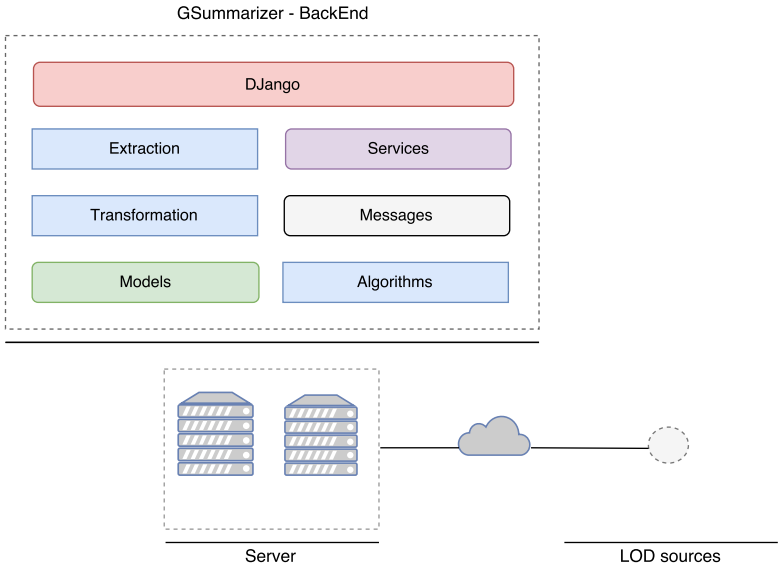


Figure 4.7: GSummarizer back-end diagram showing the high level functional modules on the architecture .

contract between the external data and the internal components of the GSummarizer. **B, E**

- **Transformation** This module exposes a set of transformation methods that can be applied to a data structure. Depending on the method , the input input requirements and output specifics can change. The input and output structures must be located in the message module. **C, F, G**

- **Algorithms** This module contains the algorithms that will be used in the transformation process or directly by the service, as the transformation module, the input and output are data structures contained in the message module. **C, F**

- **Services** This module is responsible of acting as a composer layer. This module will expose to DJango services that it can use. This usually will be a composition of extractions, transformations and algorithms. The inputs and ouputs will be explain ahead when we enter in the detail of each module.

- **Model** The model is responsible of saving the required data structures that need to be persistent onto the DB. The model in this case is

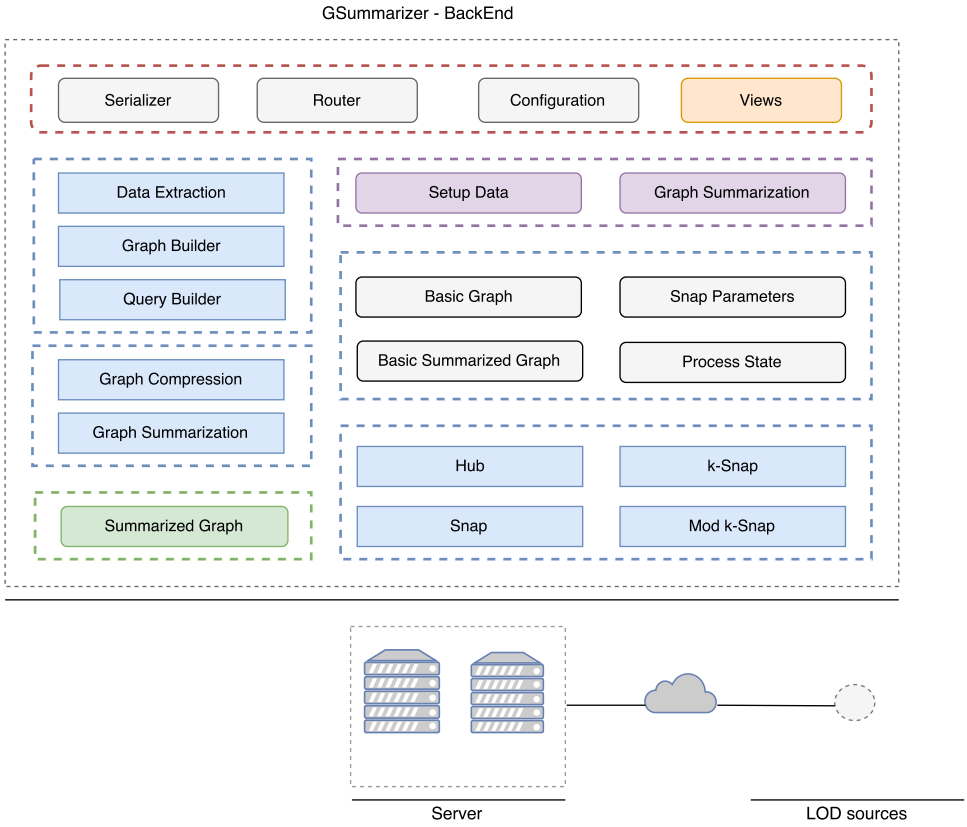


Figure 4.8: GSummarizer back-end diagram showing the current components of the each of the functional modules on the architecture .

distributed as data structures belonging to the message module in-memory and the ones saved onto the DB. For some of the applications, it may be that the model is always in-memory.

- **Messages** The message module is responsible of providing the definitions for all the data structures that will be used as input or output of the different processes in each of the modules.

In Figure 4.8, the components of the modules that are present in the current implementation of the *GSummarizer*, are displayed.

Each module in Figure 4.7 represents a set of components whose combined responsibility must enforce the module one. The global responsibility of the module is not necessarily exactly the one for each of the components of the module. From now on, when we refer to a **component** in Figure 4.8, we are referring to a hierarchy of components that have the same responsibility but

that can have different strategies to achieve it. With this we ensure to sustain the *single responsibility* principle, while making the modules *extensible* in order to comply with the initial requirements stated in Section 4.1.1.

DJango Module

The *DJango* module has four main components:

- **Serializer**, contains the structure that describes how to serialize or deserialize a data structure in order to use it as a message, serializing it in the front-end or deserializing it in the back-end.
- **Router**, contains the descriptors on how to route the calls receive from the client.
- **Configuration**, holds the descriptors on the properties of the general framework.
- **View**, is responsible of holding the process on how the view will be generated, templates, type of view, etc. In our case, all the views are *RestAPI* methods, exposed to the front-end.

Extraction Module

The *Extraction* module has three main components:

- **Data Extraction**, is responsible of getting the data form the LOD data sources. In this case uses the query builder in order to build the correct query in order to extract the data. The type of data extraction can be extended to fit different data extraction processes.
- **Graph Builder**, is responsible of transforming the input obtained from the *Data extraction* component, into a data structure compatible with the graph definitions contained in the *messages* module.
- **Query Builder** , is responsible of creating the correct query based on the input parameters and the LOD endpoint. Different endpoints can have different syntax so a set of components, one per LOD is how this family of components will be organized.

Transformation Module

The *Transformation* module has two main components:

- **Graph Compression**, is responsible of performing the post-processing operation of compressing a graph. The input of this method in our case is a *Basic Summarized Graph*, the output is a Basic Graph.
- **Graph Summarization**, is responsible to perform the graph compression processes. The input and output in our case are graphs, this graph types are defined on the *messages module*.

Algorithms Module

The *Algorithms* module has four main components:

- **Hub**, is responsible of performing the *Hub* algorithm described in the previous chapter.
- **Snap** is responsible of performing the *Adjusted Snap* algorithm described in the previous chapter.
- **k-Snap**, is responsible of performing the *Adjusted k-Snap* algorithm described in the previous chapter.
- **Mod k-Snap**, is responsible of performing the *Adjusted Snap* and *Adjusted k-Snap* combined with the *Hub* algorithm for better grouping.

Services Module

The *Services* module has two main components:

- **Setup Data**, is responsible to use the *Extraction* module in order to get the initial information that must be displayed to the user before the him/her choosing the summarization parameters in our case. In a more general case this component will get all the necessary parameters needed for the given algorithm selected by the user.
- **Graph Summarization** is responsible of using the core modules of the application and the correct components in order to provide to the user a summarization graph.

Messages Module

The *Messages* module has four main components:

- **Basic Graph**, represents the GSummarization graph data structure, presented in the previous chapter.
- **Basic Summarized Graph** represents a *Basic Graph* structure with the grouping description added.
- **Snap Parameters** represents the set of parameters needed to make the summarization, corresponding to the set P described in the previous chapter.
- **Process State**, this is a control data structure maintained through the whole process, owned by the components in the service module. While the component, execute the required components, this component records the process outcome and steps.

We have presented the static structure of the *GSummarizer* back-end. We presented the main high-level modules and the current components that satisfy the requirements stated in Section 4.1.1. The sequence of interactions are the dynamic part of the model of the back-end. Now we will explain the main sequence of interactions between the components in order to fulfill the requirements. These sequence are the continuation from the back-end perspective of the ones presented in Section 4.3, in Figures **??** and **??**.

Load Options

In Figure 4.6 the front-end requests the available options at the initial state of the application to the back-end using the *loadOptions*() method. In our case the set of options is only the set of available **LOD endpoints**. This is not the only possibility and the initial options can be extended in the future with more complex structures.

The request done by the front-end arrives to the back-end through the Rest API exposed by it with the *loadOptions*() call, this call will be manged by the service *SetupData*, this service uses the *DataExtraction* component. The data extraction will take from a list the set of available **LOD endpoints** and optionally validate checking if they're alive (ie. *isLive*() method). The result will be communicated to the front-end in the form of a serialized list. We can see the sequence diagram of this process in Figure 4.9.

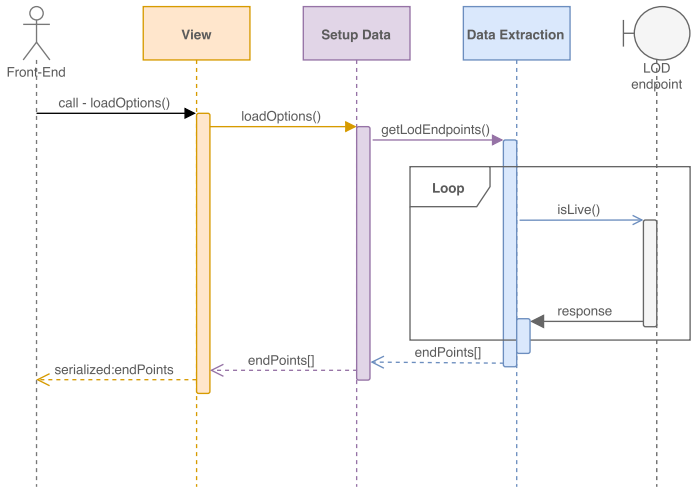


Figure 4.9: GSummarizer back-end sequence diagram representing the request of the available LOD endpoints.

Get classes, attributes and relationships

In Figure 4.6 the front-end requests to the server the available LOD endpoint set first. Afterwards, the user selects one of this endpoints and requests the set of available classes, attributes and relationships using the $getLodData(endpoint)$ method.

In the server side, the back-end will execute the call using the *SetupData* service. The service will call the *DataExtraction* component. The component will first create the right query in order to *getLodData* for the current *endpoint*. The second step will be executing the query and parsing the query result into a useful data structure that can be returned to the front-end. In this case the structure is a map of class names as *keys* and as *objects* the list of attributes and the list of relationships. At the end the map will be serialized and returned to the front-end as seen on Figure 4.10

Graph summarization

In Figure 4.5 the front-end uses the $summarization(parameters)$ call to request a summarization of the sub-graph that can be generated by the selected *parameters*. The back-end process supporting this operation is represented by Figures 4.11 and 4.12. In order to make simplify the diagram of the whole process, the diagrams was divided in two.

In Figure 4.11 the view receives the call with the parameters and dese-

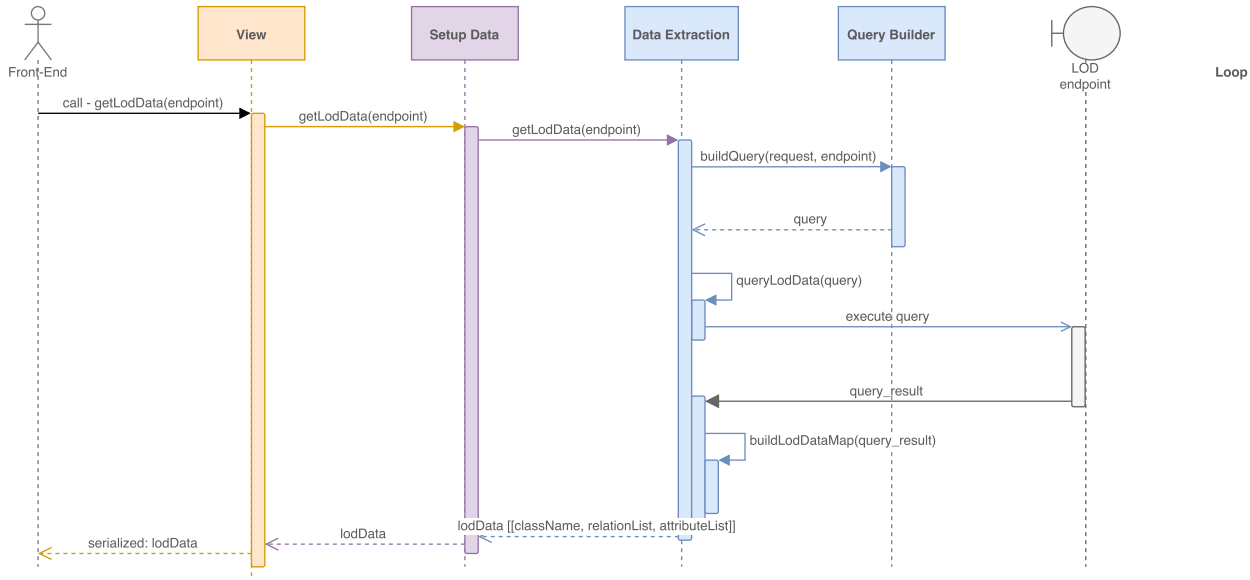


Figure 4.10: GSummarizer back-end sequence diagram representing the request of the set of classes, relationships and attributes available for the user in a given LOD dataset.

rializes it into a common structure that enforces the input expected by the back-end, the *SnapParameters*. The *SnapParameters* component belongs to the message module, which responsibility is *to enforce standard data structures that can be used among modules components.* The passed snap parameters include the type of summarization needed, this information with the deserialized parameters will be passed to the *GraphSummarizer*. The *GraphSummarizer* will first build the graph from the parameters using the *GraphBuilder*. In order to build the graph the back-end must build, using the *DataExtraction* and the *QueryBuilder*, the right query in order to obtain (1) the *instances* of the classes with the right (2) *attributes* value and (3) the list of *relatedclasses* per type of relation. After generating the query it will execute it using the *LodEndpoint* selected earlier. This data extraction component will return a data structure that can be parsed into *Nodes*, *RelationsCube* and an *AttributeMatrix*. The individual components will be joined into an structure, in this case, a *BasicGraph*. This *BasicGraph* structure will be return to the *orchestrator* in this case the service component.

In Figure 4.12 **graph summarization service component** call the **graph summarization transformation component** using the *summarization* method , passing as argument the build graph as seen in Figure 4.11. The

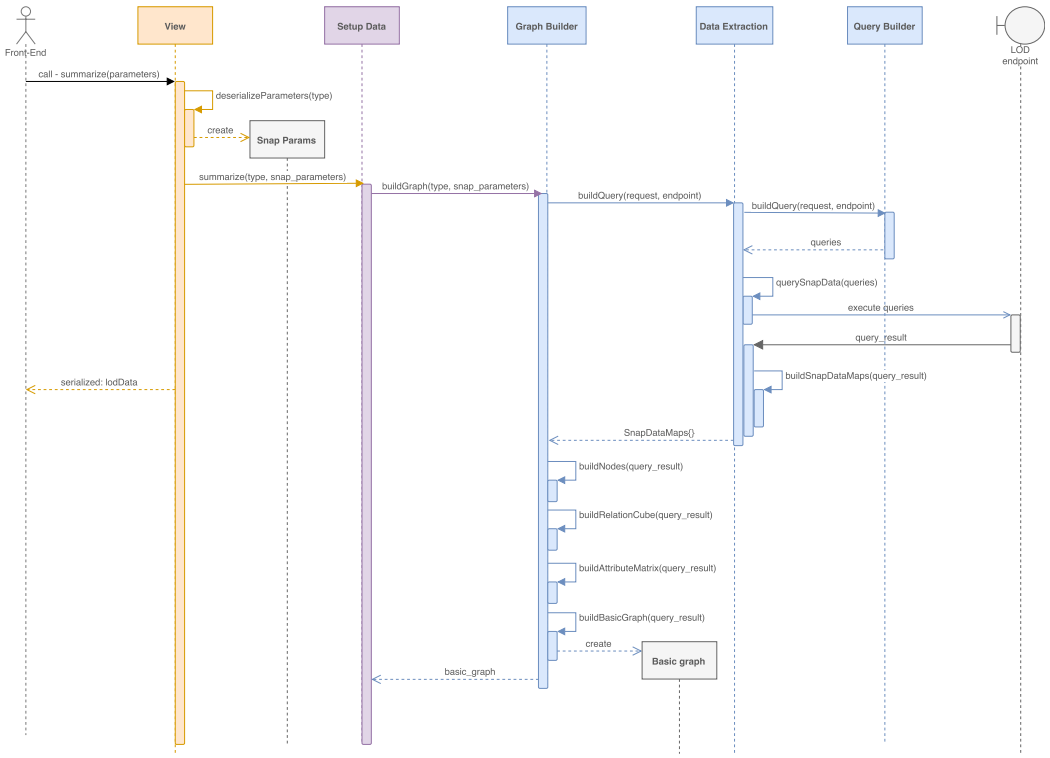


Figure 4.11: GSummarizer back-end first part of the sequence diagram representing the request of a graph summarization. This part shows how the graph is build using the LOD dataset.

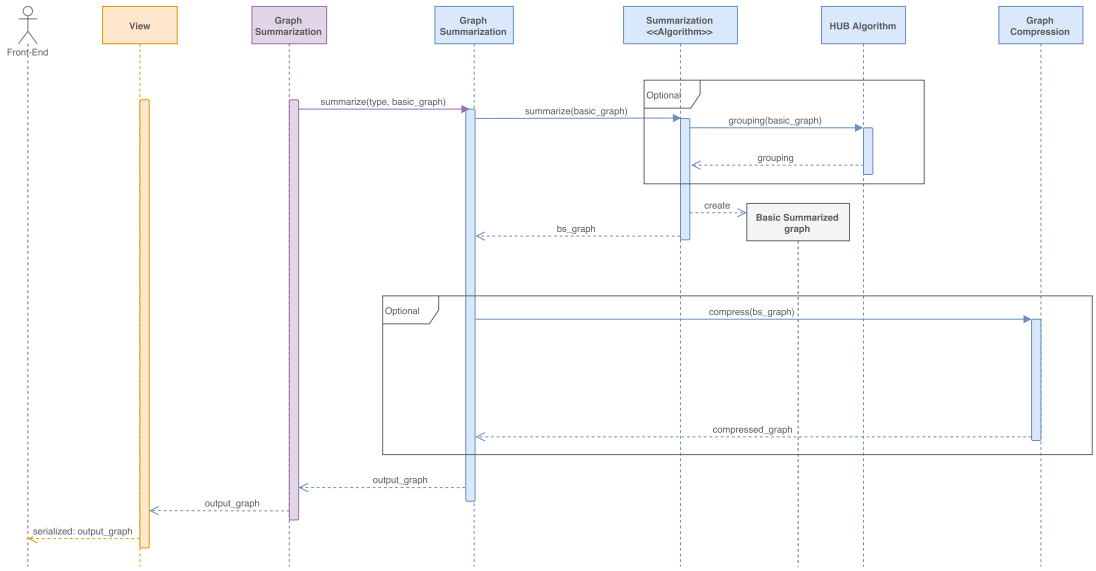


Figure 4.12: GSummarizer back-end second part of the sequence diagram representing the request of a graph summarization. This part shows how the summarization graph is build, using the obatained graph build by the previous section.

$Transformation$ component will use the **summarization algorithm component** in order to perform the operation, this component can be , in our case the component that performs the *Snap*, *k-Snap* or *Modified k-Snap* algorithms. As an optional path, the algorithms have a version using the **HUB** algorithm component. At the end of the summarization, the final grouping and the graph information will be inserted in the *Basic Summarized Graph* structure. This structure can be compressed using the **Graph Compression** component, this component will output a compressed graph, this graph has the same structure as the $BasicGraph$ in our case. As a final step it will return serialized to the front-end through the view.

Chapter 5

Experimental Analysis

In this chapter we will present the experimental results obtained after using the *GSummarizer* for testing our algorithms based on the *SNAP* techniques family. We will start by describing the experimental setup, specifying the hardware and software used. Then we will explain how the experiments were design and their individual objectives, finally we will show the experimental results of each of the test and our interpretations and analysis based on them.

5.1 Experimental setup

In this section we will present the specifics of the testing bench. We will describe the most relevant hardware elements used in the experiments and the software needed for testing the algorithms.

5.1.1 Hardware

The testing machines used for the experiments had the following hardware configuration:

- **PC:** Dell System XPS L502X
- **CPU:** Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz, AMD64 capable
- **GPU internal:** 2nd Generation Core Processor Family Integrated Graphics Controller

- **GPU external:** NVIDIA Corporation GF108M [GeForce GT 540M]
- **RAM Memory:** 8GB (2x4GiB) SODIMM DDR3 Synchronous 1333 MHz
- **Hard Drive:** Samsung SSD 840 , 250GB
- **Network:** Centrino Advanced-N 6230

The *GSummarizer* must integrate with the systems provided by the **LOD endpoints**, but these are external to the project and will not be described or taken into account for the purpose of this experiments.

Software

The technology stack for the *GSummarizer* was presented in Section 4.1.3. At the moment of the experiments, these are the versions of the most relevant software used for them:

- **OS:** Ubuntu 14.04
- **Web Server:** GUnicorn 19.6.0
- **Web Framework:** Django 2.7.12
- **Server Language:** Python 1.10
- **Front-End framework:** AngularJS 1.5.8
- **Graphical library for graphs:** VisJs 4.16.1
- **Testing application:** GSummarizer 1.0

5.2 Experimental Design

This section will present the description of the main tests that were done for the platform. The description will contain the motivation for the test and how the test will be measured and execute.

The *GSummarizer* interface provides the necessary key inputs and outputs to perform this test. All the tests contain a set of parameters in common

- *SNAP parameters*, the selected set of classes and their respective selected attributes and relations.
- *Desired number of instances*, the desired number of instances to retrieve per class.
- *Algorithm Type*, the type of algorithm (ie. A4)

The *SNAP Parameters* are composed of n different classes up to a maximum of 5 classes, but per class only one combination of relations and attributes will be used in order to reestrict the number of possible combinations for the tests.

The *desired number of instances* p, creates an equivalent number of n instances per class, therefore the number of nodes is at least $p \times |selectedclasses| + |relatedclassesinstances|$. The problem with this approach is that the data structures depend on the number of relations and attributes chosen. The data structure that grows exponentially with the number of relation is the *RelationshipCube*. In order to make a fair comparison among tests by the number of nodes we simplify the effective number of nodes as:

$$nodes_{effective} = n_{final} \times |\Gamma| \tag{5.1}$$

where n_{final} is the final number of nodes in the output raw graph and Γ is the set of relationships.

The **algorithm types** that will be used in all the tests are the following:

A1 **Raw:** The raw graph

A2 **SNAP only attributes:** The summarization using only the grouping by attributes method.

A3 **SNAP:** The modifed snap algorithm

A4 **kSNAP:** The modified k-snap algorithm

A5 **SNAP only attributes + HUB:** The summarization using the grouping by attributes and then the grouping using the hub algorihtm.

A6 **HUB + SNAP:** The modified snap algorithm with the hub grouping.

A7 **HUB + kSNAP:** The modified k-snap algorithm with the hub grouping.

A8 **HUB + SNAP + Compression:** The compression of the modified snap algorithm with the hub grouping.

A9 **HUB + kSNAP + Compression:** The modified k-snap algorithm with the hub grouping.

The types of algorithms are combinations or single applications of the algorithms described in Chapter 3.

5.2.1 Nodes vs Execution time

The execution time is a very important measure to determine how well a system performs. In this case a user must interact with the system to achieve a result and as such the *execution time* mus be studied. The idea of this test is to determine the behaviour of the algorithms under different work loads. The sets in this case will be composed of $[1, 2, 3]$ classes with maximum 20 attributes overall and maximum of 6 relationships overall. From previously run tests, the results yield that over these number the execution time results go far beyond the usability spectrum for a web user.The desired number of instances will be $[20, 200, 600, 1000]$ these numbers are bounded by the same logic as the previous one, increase times beyond these numbers are not usable in the context of this study. Finally the execution time is the sum of the *build time* for the raw graph, *summarization time* and in the cases that is relevant, the *compression time*.

5.2.2 Nodes vs Grouping

One of the objectives of using a summarization technique, is to reduce the size of the graph into a significantly more usable one. In the case of our study , the relevance is a subjective measure, but how well it reduce the size of the node based on an algorithm that produce relevant summarizations is an indirect way to measure relevancy **??**. Therefore this test measures the initial node size vs how many groups were form after applying the algorithm. In an indirect way, the smaller the grouping is, it is probable that the summarization is better. The experimental configuration is the same as the one presented in Section 5.2.1

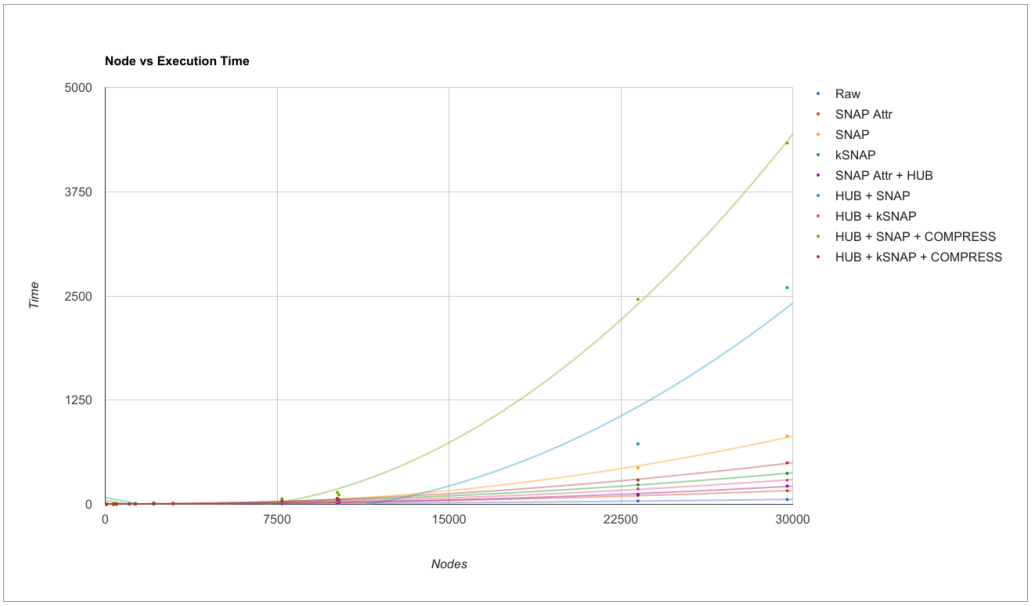


Figure 5.1: Number of nodes vs. Execution time for all the algorithm techniques

5.2.3 Visualization Comparison

The objective in this experiments is to compare the different algorithms when visualized in the front end while changing the parameters mentioned at the beginning of this section. In order to do that, the variables will be used accordingly to the experimental variations presented in Section 5.2.1.

5.3 Experiments and Analysis

In this section we will present the main results of the experiments done in the *GSummarizer* platform.

5.3.1 Nodes vs Execution time

In Figure's 5.1 scale we can observe that most of the algorithms curves are below $1000 seconds$ or $16 minutes$, but the techniques that increase the exe-

cution time exponentially are based on the SNAP algorithm, specifically the *SNAP+HUB* and the *SNAP+HUB+COMPRESSED* algorithm. The exponential growth in time of these algorithms is due to two factors, first both will compute groupings until it arrives to $\Phi_{A,R}^{max}$ grouping, unlike the *kSnap* that will begin with the Φ_{A}^{max} and iteratively will drill-down to reach the k number. The second factor that affects these techniques is the use of the Hub algorithms. The Hub algorithm will increase the size of the Φ_{A}^{max} grouping. The increase in size is necessary so the initial number of groupings is not so low because of the *incompleteness of attribute data* problem explained in Section 3.4.3. Therefore a bigger initial grouping means that it will take more time for the *Snap* algorithm to reach $\Phi_{A,R}^{max}$. From this , all the techniques that will use $SNAP + HUB$ will be affected by these phenomenon.

As stated in Chapter 3.4.3 the incomplete data problem affects the Φ grouping, causing a large initial grouping Φ_{A}^{max}. The effect of a large initial grouping in the case of the $kSnap - based$ techniques is that the k number will not affect the grouping in a significant way. In **??** they show that the number k that scales more linearly is $k = 10$, therefore we use this number in all the tests given that the output will not be affected significantly.

In Figure 5.2 we trim the curves that go above the $1000 seconds$ for the maximum number of nodes. In the curves presented on this Figure we see that the technique that took the longest as expected is the *Snap* but still it was faster than any of the $Snap + HUB - based$ techniques. This is caused because the initial grouping Φ_{A}^{max} contains a lot of groups so it is closer to the $\Phi_{A,R}^{max}$, than the ones using the Hub algorithm. The Hub algorithm create a coarser grouping, leaving the algorithm to increase the number of iterations in order to achieve $\Phi_{A,R}^{max}$, than with the Hub grouping, the generated information will be greater.

The algorithms that took the less time where the baselines or references algorithms, as expected the RAW, *SnapAttribute* and the $Snapattribute + Hub$. The RAW algorithms is the main baseline given that it is the graph information as-is. The other two algorithms create the Φ_{A}^{max} and the latter creates a more coarse and information-rich grouping thanks to the Hub algorithm. All three are the baselines for the subsequent algorithms so as expected they have the lows latency.

The best algorithm with the lowest latency after the baselines is the $kSnap + HUB$ one. The main reason is that the baseline is coarser, the modified Φ_{A}^{max} contains information of the biggest relationship hubs as de-

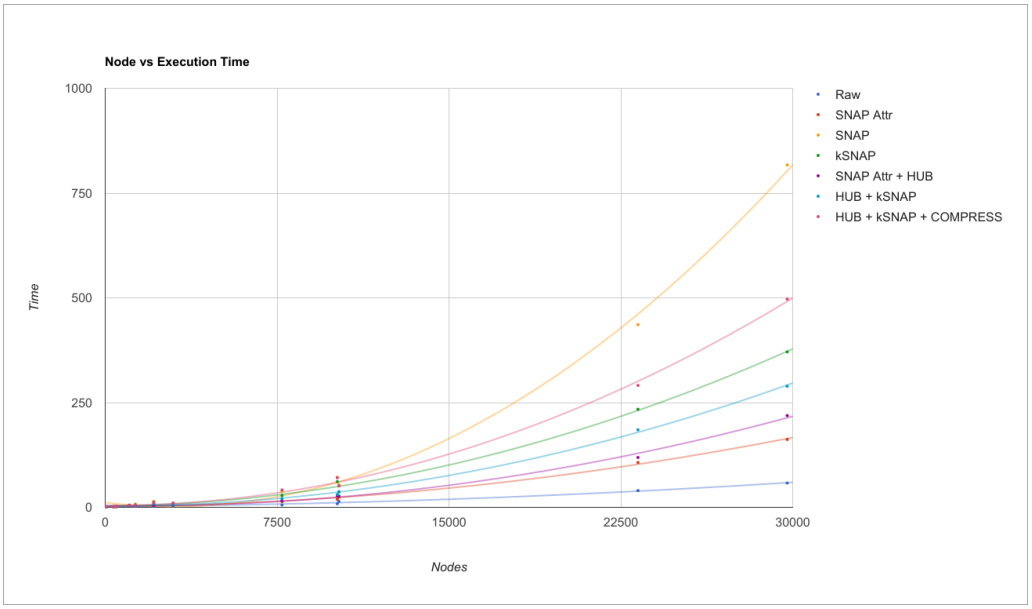


Figure 5.2: Number of nodes vs. Execution time for all the algorithm techniques

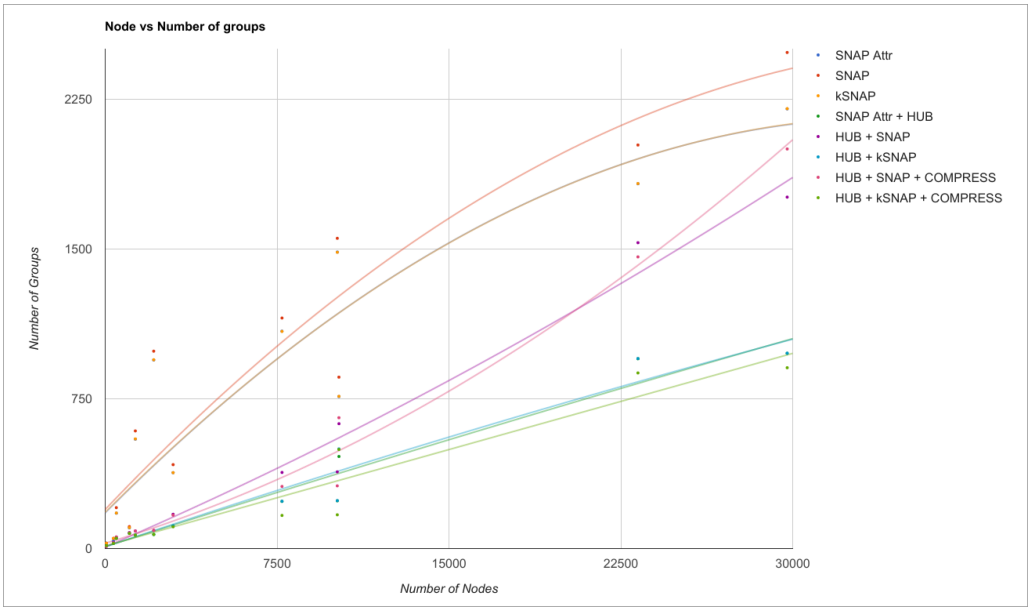


Figure 5.3: Nodes vs. Grouping techniques

scribed in section 3.4.4. A coarser initial grouping means, less groups to iterate over in order to calculate the next partition.

From Figures 5.1 and 5.2 , we can see that the *kSnap* algorithm combined with the *Hub* have better performance in execution time but still we need to check how is the grouping capabilities behind each of the algorithms.

5.3.2 Nodes vs Grouping

In Figure 5.3 we can observe the output of groups based on the input of nodes per algorithm. This is a way of measuring the capacity of the algorithm to summarize the node. We are not giving a qualification to the summarization but instead we are using the grouping output as a quality measure because at the end a smaller number of groups is better for the human eye in any case. The *Snap* and the *kSnap* algorithms are the ones that performs the worse, this is due to the big initial grouping problem mentioned earlier in this chapter. If the initial size of Φ_A^{max} is big, the output size of the grouping

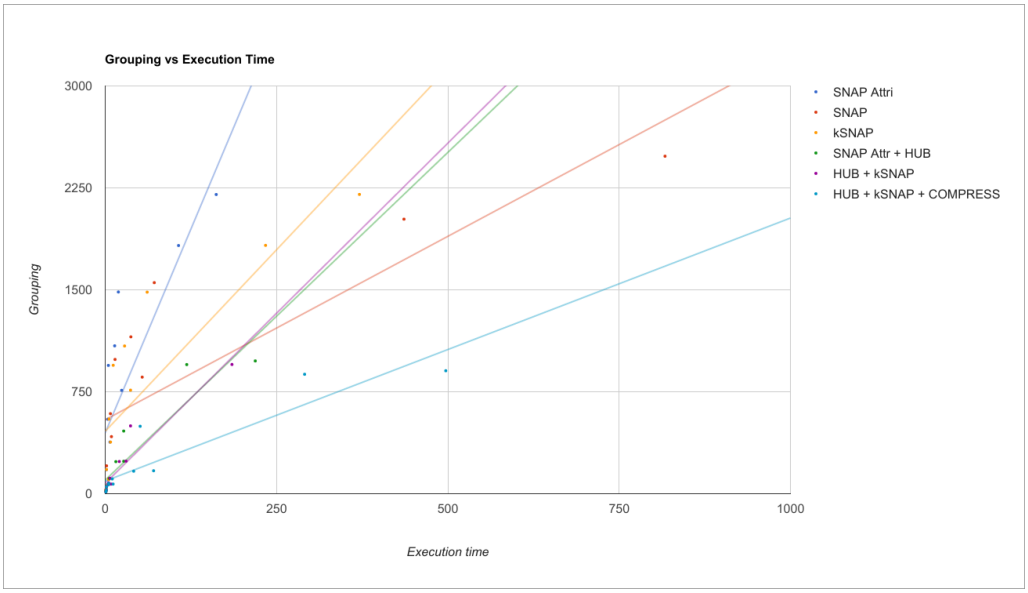


Figure 5.4: Grouping vs. Execution time

cannot be small. The initial grouping size is aid by the HUB algorithm, as such, the best performant techniques are the ones based on the $kSnap+HUB$ algorithm. From the best performant techniques we can observe a linear behaviour in the increase of input nodes and the output grouping size.

The idea of a good algorithm in this case will be one that in short time will create the smaller increase in grouping size. Is important not only that the grouping is small in a certain point of time but that the size increment over time is smaller because that would mean that the the increase of information doesn't increase as much the summarization, meaning that the earlier version of the summarizations already capture the main conceptual peaces of information from the graph. In Figure 5.4 we can see that the algorithm that have the lowest inclination is the one from the $Hub+kSnap+COMPRESS$ technique. This means that the number of groupings will increase in a more controlled way than in other cases, therefore the summarization will be more scalable, given that it will not explode the number of groupings

like in the case of $SNAPAttr$. In the case of $kSnap$, it behave well in $NodesvsExecutionTime$, but now that is measured against the grouping size , it escalates poorly.

At the end of the test depicted in figures 5.1, 5.2, 5.3 and 5.4 we see that the best performant algorithm for used test is the $Hub + kSnap + COMPRESS$. The algorithm performs better given that the Hub algorithms helps to create a better and more coarsed initial grouping, making the following iterative steps for the $kSnap$ a less time consuming task and finally it creates less groups by compressing the nodes, pruning the ones that do not increase the information content of the summarization.

5.3.3 Visual grouping Comparison

In this section we will present the visual representations of the summarization performed by the $GSummarizer$ for a single set of parameters. This is an illustrative set of tests in order to show the correspondence between the results obtained in the previous section and the visual output of the tests. In order to study the usefulness of the visualization is out of the scope of this work and will be left for future studies. The tests where performed using a fix set of input parameters P and recording the output for each of the algorithms used in the previous sections.

In Figure 5.5 we have the raw graph of a request with an output of 3000 nodes. The nodes are represented by the colored circles, the raw visualization display the nodes as blue only. We can see several groupings of nodes, towards the center the most numerous ones, bear in mind that this is the raw graph without any kind of grouping applied to it.

In Figures 5.6a and 5.6b we can see a summarization based on the $attribute-only$ grouping algorithm, therefore the clusters that we can see are the ones belonging to Φ_A^{max} output. The clusters in this case are represented by blue squares and the nodes as colored circles. In the summarization on 5.6a we can observed some clusters and nodes attached to them, but the density of clusters is much lower compared to the ones on 5.6a. The contrast between the two graph outputs is as expected from the two algorithms and coherent with the previous findings, where the Hub algorithm enhances the grouping process creating a new layer of clusters.

Now lets compare the two figures above: 5.7a and 5.7b, the $kSnap$ generates more clusters given that the summarization is coarser. The coarser

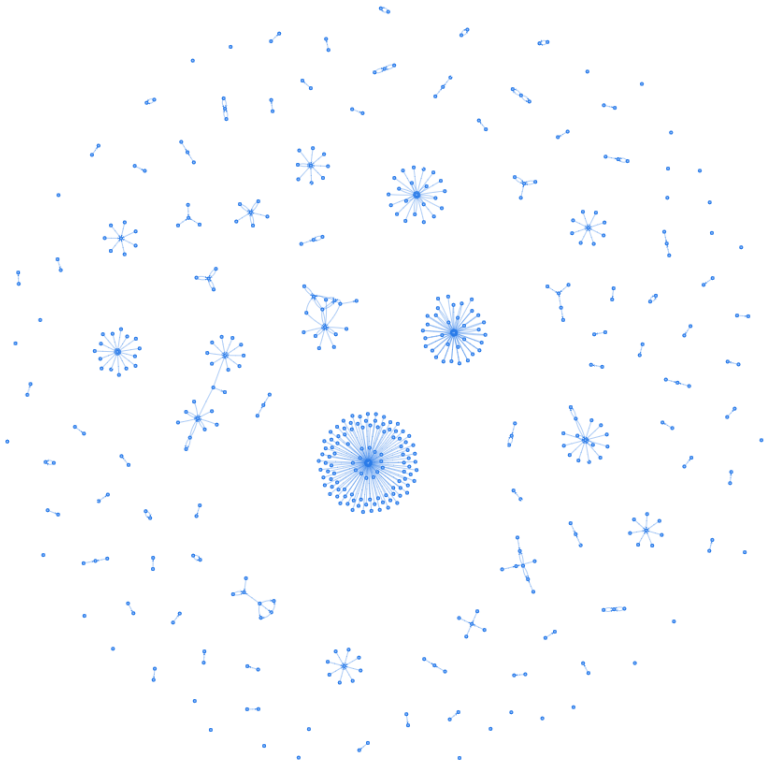
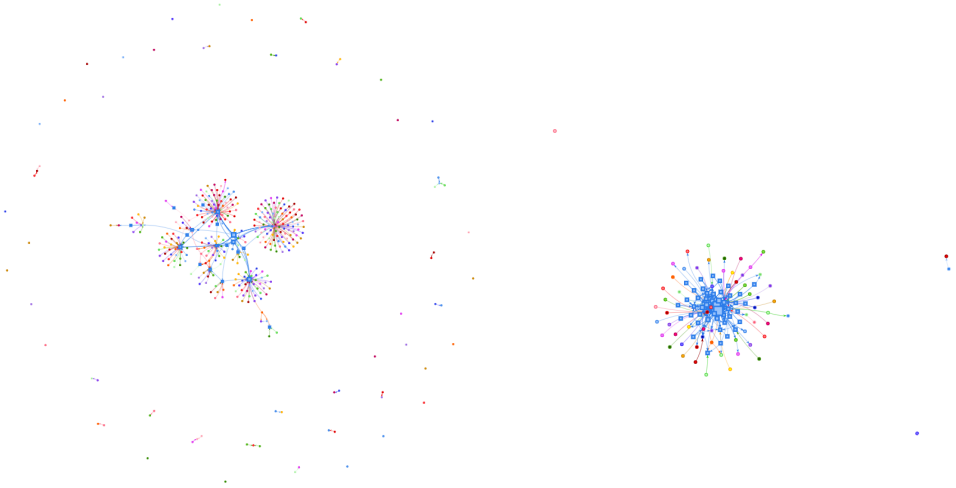
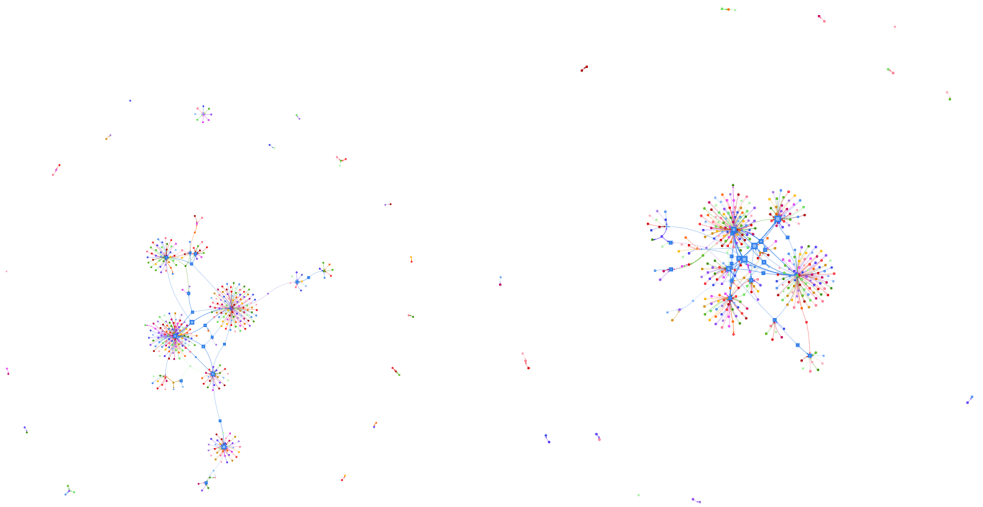


Figure 5.5: Raw graph



(a) Snap Attributes *(b) Snap Attributes + Hub*

Figure 5.6: The initial grouping algorithms



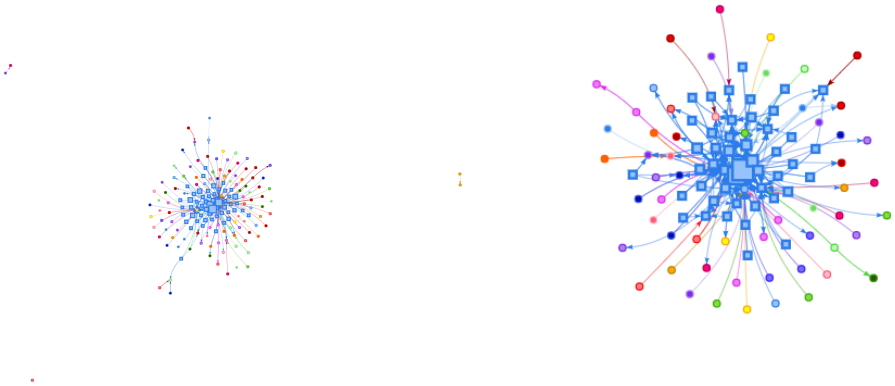
(a) Snap (b) kSnap

Figure 5.7: Comparisson of the $kSnap$ and the $SNAP$ algorithm

summarization is as expected given that the *kSnap* dont reach the $\Phi_{A,R}^{max}$ grouping, but stops before in the usual case. The *kSnap* then will have more clusters and more numerous than the *Snap* summarization.

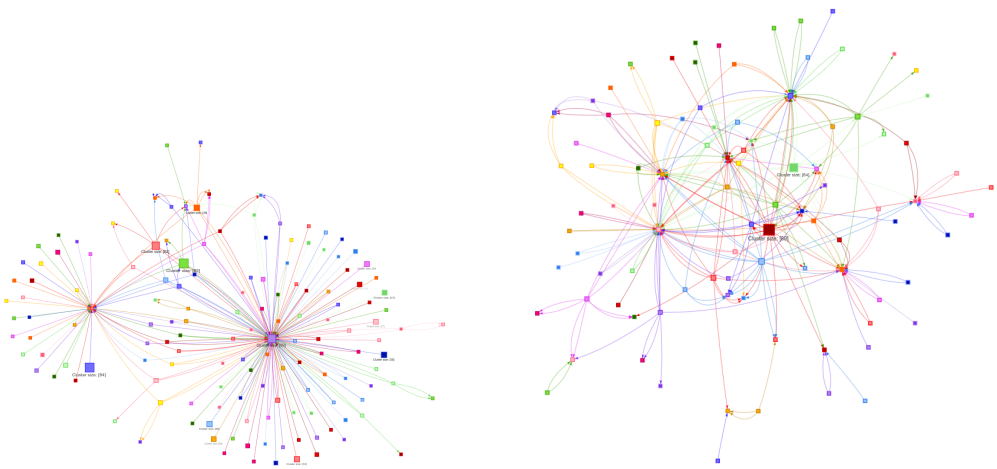
The *Hub* algorithm provides an coarser initial grouping on top of the Φ_{A}^{max} one. An initial coarser grouping will lead to a coarser final grouping in the case of the *Snap* (Figure 5.8a) and the *kSnap* (Figure 5.8b) algorithms. In the Figures they seem similar but in reality, as we have seen in the previous graphs, the latter algorithm creates a much coarser grouping in any case.

The compressed algorithms create a different kind of visualization in this case. When the summarization is compressed, all nodes are in reality clusters, therefore are shaped as squares. The bigger the cluster, the bigger the squares. We can see in Figures 5.9a and 5.9b that usually the central cluster wil be the biggest one. The central cluster in Figure 5.10 has 89 nodes within it. We define how heavy a graphical summarization is by the *quantity of elements displayed*. Both techniques shows a great increase in cluster quantity and contain less visual weight than the previous visualization. The compressed *kSnap* combine with *Hub* still delivers the *less graphically heavy* of all the summarizations and this corresponds perfectly with the results obtained in the previous sections.



(a) Snap + Hub *(b) kSnap + Hub*

Figure 5.8: The comparisson of the main algorithms combined with the Hub one.



(a) Snap+Hub+Compressed *(b) kSnap+Hub+Compressed*

Figure 5.9: The compressed algorithms

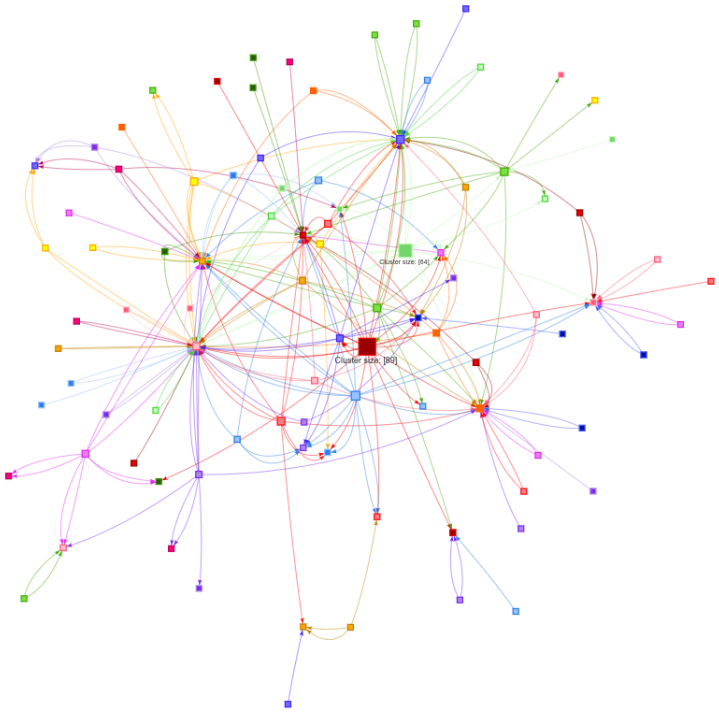


Figure 5.10: kSnap+Hub+Compressed detail

Chapter 6

Conclusions

This is the final chapter of this work. In this chapter we will present first our conclusions after working on this research and then we will show the suggestions to improve upon the current state of this investigation as a future work section.

1. Within the set of modified *Snap family* of algorithms used in this work, the one that performed the best according to *execution time and grouping number* was the compressed, combined *kSnap, Hub* one. As seen in Chapter 5, the advantages of this algorithm was the initial big grouping, contained iterative cycles for creating new groupings and cleaner output. The initial big grouping was provided by the Hub algorithm, that augment the information and group size by clustering using the one-hop relations. The kSnap provide the shorter iterative cycles than the SNAP algorithm. The post processing or compressed algorithm provide a better output graph removing useless information.

2. The resulting graph generated by the compressed algorithms presented in this work represent implicitly a weighted network. The nodes in the compressed graph represent clusters and each cluster have a value, in this case the number of clusters. The previous weight can be combined with the number of connections with other weighted nodes. This kind of networks can be mine again for more information, therefore making the output of this algorithms a possible source for other link mining algorithms as a pre-processing scheme.

3. Real *Linked Open Data* like many real data sets contained a lot of missing information. Other problem presented is the duplication of data,

specifically with equivalent relationships. The LOD problems when summarizing a graph were partially solved modifying using methods like *Hub* and compressing the graph as a post-processing practice. But from the experimental results in Chapter 5, the algorithms here create a better base for a further summarization than being a clear summarization by itself.

4. In order generate better summarizations from the LOD data set would be advisable if not necessary to use two or more hops extraction given that relationships in the graph provides more information between classes of objects. Better relationship information, makes the relationship grouping algorithms like Hub to generate bigger and better groups, with objects that are more related to each other, therefor aiding the summarization methods that will use the initial grouping as base. In this are , this work was limited by the restriction of getting only one-hop nodes based on the relationship given by the user.

6.1 Future work

In this section we will present suggestion on how future researchers could build upon the work done in the investigation presented in this document.

1. In order to increase the amount of information available to present a better graph summarization, a *n-hop extraction* should be added to the process of getting the raw graph data. In order to obtain a n-hop extraction it would be necessary to have a second set of parameters that specify, from the *related node classes*, the relationships that will be used to extract the 2-hop related nodes or second level nodes. In a general case we should have n-sets of related nodes starting from the one-hop until the n-hop. One of the challenges of having this n-hop extraction is to define the second set of relationships. To ask to the user a second set of relationships is inconvenient, because , the user is seeking advice to explore and visualize the LOD data set and we cannot scale this process easily given that the user will have to input for n-sets of relationships. A future enhancement to the work presenting in this document and related works in the field of graph summarization and link mining would be to search on how to automatically decide on the

best possible second set of relationships in order to extract the two-hop instances and if it is possible and relevant determine how to do it for $n - hops$.

2. The compressed $kSnapplusHub$ algorithm provides a summarization that groups the nodes based on their relationships and attributes, and the compression clean the summarized graph from nodes that increase the sparsity of the matrices describing the summarized graph. While the summarization creates identifiable clusters that can be explored by the user, if the requested initial amount of instances is big (with respect to the numbers use in this study) it can increase the size of the graph to a point that is still not easily readable for a user. In order to create a more readable summarization, a future enhancement would be to use other algorithms on top of this summarization, therefore use it as a pre-processing method in order to build upon the already summarize data. A suggested method would be one that can create human-readable inferences from a graph.

Bibliography

[1] SÃ¶ren Auer et al. “Introduction to Linked Data and Its Lifecycle on the Web”. In: *Reasoning Web. Semantic Technologies for Intelligent Data Access Lecture Notes in Computer Science* (2013), 1â90. DOI: 10.1007/978-3-642-39784-4_1.

[2] *Best Practice Recipes for Publishing RDF Vocabularies*. URL: http://dret.net/biblio/reference/swbpvocabpub.

[3] Christian Bizer, Tom Heath, and Tim Berners-Lee. “Linked Data”. In: *Semantic Services, Interoperability and Web Applications* (), 205â227. DOI: 10.4018/978-1-60960-593-3.ch008.

[4] “Browsing Linked Data with Fenfire”. In: *Linked Data on the Web (LDOW2008)*. URL: http://data.semanticweb.org/workshop/LDOW/2008/paper/19.

[5] Josep Maria Brunetti et al. “Formal Linked Data Visualization Model”. In: *Proceedings of International Conference on Information Integration and Web-based Applications Services - IIWAS '13* (2013). DOI: 10.1145/2539150.2539162.

[6] Diego Valerio Camarda, Silvia Mazzini, and Alessandro Antonuccio. “LodLive, exploring the web of data”. In: *Proceedings of the 8th International Conference on Semantic Systems - I-SEMANTICS '12* (2012). DOI: 10.1145/2362499.2362532.

[7] Rathachai Chawuthai and Hideaki Takeda. “RDF Graph Visualization by Interpreting Linked Data as Knowledge”. In: *Semantic Technology Lecture Notes in Computer Science* (2016), 23â39.

[8] E.h. Chi. “A taxonomy of visualization techniques using the data state reference model”. In: *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings* (). DOI: 10.1109/infvis.2000.885092.

[9] Dadzie and Rowe. “Approaches to visualizing linked data: a survey”. In: *Semantic Web* 2.2 (2011), 89â124. DOI: 10.3233/SW-2011-0037.

[10] Pedro Domingos et al. “Markov Logic: A Language and Algorithms for Link Mining”. In: *Link Mining: Models, Algorithms, and Applications* (2010), 135â161. DOI: 10.1007/978-1-4419-6515-8_5.

[11] Marc Downie et al. “Evolving a rapid prototyping environment for visually and analytically exploring large-scale Linked Open Data”. In: *2011 IEEE Symposium on Large Data Analysis and Visualization* (2011). DOI: 10.1109/ldav.2011.6092338.

[12] Cody Dunne and Ben Shneiderman. “Motif simplification”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13* (2013). DOI: 10.1145/2470654.2466444.

[13] Nicola Fanizzi, Claudia Damato, and Floriana Esposito. “Mining Linked Open Data through Semi-supervised Learning Methods Based on Self-Training”. In: *2012 IEEE Sixth International Conference on Semantic Computing* (2012). DOI: 10.1109/icsc.2012.54.

[14] Flavius Frasincar, Alexandru Telea, and Geert-Jan Houben. “Adapting Graph Visualization Techniques for the Visualization of RDF Data”. In: *Visualizing the Semantic Web* (), 154â171. DOI: 10.1007/1-84628-290-x_9.

[15] Lise Getoor. “Link mining”. In: *ACM SIGKDD Explorations Newsletter* 5.1 (2003), p. 84. DOI: 10.1145/959242.959253.

[16] Lise Getoor and Christopher P. Diehl. “Link mining”. In: *ACM SIGKDD Explorations Newsletter* 7.2 (2005), 3â12. DOI: 10.1145/1117454.1117456.

[17] Hugh Glaser and Harry Halpin. “The Linked Data Strategy for Global Identity”. In: *IEEE Internet Computing* 16.2 (2012), 68â71. DOI: 10.1109/mic.2012.39.

[18] Olaf Hartig and Johann-Christoph Freytag. “Foundations of traversal based query execution over linked data”. In: *Proceedings of the 23rd ACM conference on Hypertext and social media - HT ’12* (2012). DOI: 10.1145/2309996.2310005.

[19] Michael Hausenblas. “Exploiting Linked Data to Build Web Applications”. In: *IEEE Internet Computing* 13.4 (2009), 68â73. DOI: 10.1109/mic.2009.79.

[20] Tom Heath. “Linked Data - Welcome to the Data Network”. In: *IEEE Internet Computing* 15.6 (2011), 70â73. DOI: 10.1109/mic.2011.153.

[21] Tom Heath and Christian Bizer. “Linked Data: Evolving the Web into a Global Data Space”. In: *Synthesis Lectures on the Semantic Web: Theory and Technology* 1.1 (2011), 1â136. DOI: 10.2200/s00334ed1v01y201102wbe001.

[22] Keith Henderson et al. “It’s who you know”. In: *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’11* (2011). DOI: 10.1145/2020408.2020512.

[23] Keith Henderson et al. “RolX”. In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’12* (2012). DOI: 10.1145/2339530.2339723.

[24] G. E. Hinton. “Reducing the Dimensionality of Data with Neural Networks”. In: *Science* 313.5786 (2006), 504â507. DOI: 10.1126/science.1127647.

[25] Karwan Jacksi, Nazife Dimililer, and Subhi R. “State of the Art Exploration Systems for Linked Data: A Review”. In: *International Journal of Advanced Computer Science and Applications* 7.11 (2016). DOI: 10.14569/ijacsa.2016.071120.

[26] Dr Eldhose T John, Bibu Skaria, and P.x. Shajan. “An Overview of Web Content Mining Tools”. In: *Bonfring International Journal of Data Mining* 6.1 (2016), 01â03. DOI: 10.9756/bijdm.8126.

[27] Roman Kontchakov, Mariano RodrÃguez-Muro, and Michael Zakharyaschev. “Ontology-Based Data Access with Databases: A Short Course”. In: *Reasoning Web. Semantic Technologies for Intelligent Data Access Lecture Notes in Computer Science* (2013), 194â229. DOI: 10.1007/978-3-642-39784-4_5.

[28] Dr. Sanjay Kumar. “A Comparative Study of Various Data Transformation Techniques in Data Mining”. In: *International Journal of Scientific Engineering and Technology* 4.3 (2015), 146â148. DOI: 10.17950/ijset/v4s3/305.

[29] *Latest "RDF Primer" versions*. URL: http://www.w3.org/TR/rdf-primer/.

[30] Angela Lausch, Andreas Schmidt, and Lutz Tischendorf. “Data mining and linked open data â New perspectives for data analysis in environmental research”. In: *Ecological Modelling* 295 (2015), 5â17. DOI: 10.1016/j.ecolmodel.2014.09.018.

[31] Kang Li et al. “LRBM: A Restricted Boltzmann Machine Based Approach for Representation Learning on Linked Data”. In: *2014 IEEE International Conference on Data Mining* (2014). DOI: 10.1109/icdm.2014.22.

[32] M Mathieu, H Sebastien, and J Mathieu. “Gephi: an open source software for exploring and manipulating networks”. In: *AAAI* (2009).

[33] M. Mccord and M. Chuah. “Spam Detection on Twitter Using Traditional Classifiers”. In: *Lecture Notes in Computer Science Autonomic and Trusted Computing* (2011), 175â186. DOI: 10.1007/978-3-642-23496-5_13.

[34] Aditya Krishna Menon and Charles Elkan. “Link Prediction via Matrix Factorization”. In: *Machine Learning and Knowledge Discovery in Databases Lecture Notes in Computer Science* (2011), 437â452. DOI: 10.1007/978-3-642-23783-6_28.

[35] AndrÃ¡s Micsik, ZoltÃ¡n TÃ³*th*, *AndSÃ*!‘*ndorTurbucz*. “LODmilla: Shared Visualization of Linked Open Data”. In: *Communications in Computer and Information Science Theory and Practice of Digital Libraries – TPDL 2013 Selected Workshops* (2014), 89â100. DOI: 10.1007/978-3-319-08425-1_9.

[36] Aswini Kumar Mohanty, Manas Ranjan Senapati, and Saroj Kumar Lenka. “A novel image mining technique for classification of mammograms using hybrid feature selection”. In: *Neural Computing and Applications* 22.6 (2012), 1151â1161. DOI: 10.1007/s00521-012-0881-x.

[37] Axel-Cyrille Ngonga Ngomo et al. “Introduction to Linked Data and Its Lifecycle on the Web”. In: *Reasoning Web. Reasoning on the Web in the Big Data Era Lecture Notes in Computer Science* (2014), 1â99. DOI: 10.1007/978-3-319-10587-1_1.

[38] Lixian Ni et al. “Visualizing Linked Data with JavaScript”. In: *2013 10th Web Information System and Application Conference* (2013). DOI: 10.1109/wisa.2013.48.

[39] *OWL 2 Web Ontology Language Primer (Second Edition)*. URL: http://www.w3.org/TR/owl2-primer/.

[40] *OWL Web Ontology Language Overview*. URL: https://www.w3.org/TR/owl-features/.

[41] “PageRank Algorithm, 1998; Brin, Page”. In: *SpringerReference* (). DOI: 10.1007/springerreference_57796.

[42] Axel Polleres et al. “RDFS and OWL Reasoning for Linked Data”. In: *Reasoning Web. Semantic Technologies for Intelligent Data Access Lecture Notes in Computer Science* (2013), 91â149. DOI: 10.1007/978-3-642-39784-4_2.

[43] A. Johannes Pretorius and Jarke J. Van Wijk. “What Does the User Want to See? What do the Data Want to Be?” In: *Information Visualization* 8.3 (2009), 153â166. DOI: 10.1057/ivs.2009.13.

[44] *RDF 1.1 Concepts and Abstract Syntax*. URL: http://www.w3.org/TR/rdf11-concepts/.

[45] *RDF Vocabulary Description Language 1.0: RDF Schema*. URL: https://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[46] *RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax*. URL: http://tools.ietf.org/html/rfc3986.

[47] Percy E. Rivera Salas et al. “Publishing Statistical Data on the Web”. In: *2012 IEEE Sixth International Conference on Semantic Computing* (2012). DOI: 10.1109/icsc.2012.16.

[48] N. Shadbolt, T. Berners-Lee, and W. Hall. “The Semantic Web Revisited”. In: *IEEE Intelligent Systems* 21.3 (2006), 96â101. DOI: 10.1109/mis.2006.62.

[49] Shashi Shekhar, Pusheng Zhang, and Yan Huang. “Spatial Data Mining”. In: *Data Mining and Knowledge Discovery Handbook* (2009), 837â854. DOI: 10.1007/978-0-387-09823-4_43.

[50] Martin G. Skjaveland. “Sgvizler: A JavaScript Wrapper for Easy Visualization of SPARQL Result Sets”. In: *Lecture Notes in Computer Science The Semantic Web: ESWC 2012 Satellite Events* (2015), 361â365. DOI: 10.1007/978-3-662-46641-4_27.

[51] *SPARQL 1.1 Query Language*. URL: http://www.w3.org/TR/sparql11-query/.

[52] *SPARQL Query Language for RDF*. URL: http://www.w3.org/TR/rdf-sparql-query/.

[53] *SweoIG/TaskForces/CommunityProjects/LinkingOpenData*. URL: https://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData#Project_Description.

[54] Lei Tang and Huan Liu. “Relational learning via latent social dimensions”. In: *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’09* (2009). DOI: 10.1145/1557019.1557109.

[55] Yuanyuan Tian and Jignesh M. Patel. “Interactive Graph Summarization”. In: *Link Mining: Models, Algorithms, and Applications* (2010), 389â409. DOI: 10.1007/978-1-4419-6515-8_15.

[56] “Weaving the Web: the original design and ultimate destiny of the World Wide Web by its inventor”. In: *Choice Reviews Online* 37.07 (2000). DOI: 10.5860/choice.37-3934.

[57] *What Is a URI and Why Does It Matter?* URL: http://www.ariadne.ac.uk/issue65/thompson-hs.

[58] Å ejla ÄebiriÄ, FranÃ§ois GoasdouÃ©, and Ioana Manolescu. “Query-oriented summarization of RDF graphs”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), 2012â2015. DOI: 10.14778/2824032.2824124.

Appendix A

Platform specific documentation

This chapter present the most important technology specific implementation details mentioned throughout this document.

A.1 Data Extraction Module

A.1.1 Query Builder

Get all classes query builder method

```
def dbpedia_get_all_classes(self):

        query = """
            PREFIX dbpedia-owl:<http://dbpedia.org/ontology
               />
            PREFIX dbp:<http://dbpedia.org/property/>
            PREFIX dbo:<http://dbpedia.org/ontology/>
            PREFIX owl:<http://www.w3.org/2002/07/owl#/>
            SELECT DISTINCT ?class_type
            WHERE {
                    ?class rdf:type ?class_type.
            } LIMIT 1000
            """

        return query
```

Get all relations ffrom a class

```
def dbpedia_get_all_relations(self, class_name):

        query = """
            PREFIX dbpedia-owl:<http://dbpedia.org/ontology
                />
            PREFIX dbp:<http://dbpedia.org/property/>
            PREFIX dbo:<http://dbpedia.org/ontology/>
            PREFIX owl:<http://www.w3.org/2002/07/owl#/>
            SELECT DISTINCT ?relation_type
            WHERE {
                    ?class rdf:type <""" + class_name + """
                        >;
                            ?relation_type ?object .

                    FILTER(!isLiteral(?object))

            } LIMIT 1000
            """
        return query
```

Get all instances from a given class with given relationships

```
def dbpedia_get_class_subject_instances(self, class_list,
    relation_list, instance_number):

        # 1. Build the class list for the IN clause
        in_string = ''

        for class_url in class_list:
            in_string += '<' + str(class_url) + '>,'

        # -- Remove last character
        in_string = in_string[:len(in_string) - 1]

        # --debug
        # print '\n\nThe class url list for the query is: '
        # print in_string

        # 2. Build the relation list for the IN clause
        in_relation_string = ''
```

```
        for relation_url in relation_list:
            in_relation_string += '<' + str(relation_url) +
                '>,'

        # -- Remove last character
        in_relation_string = in_relation_string[:len(
            in_relation_string) - 1]

        # --debug
        # print '\n\nThe relation url list for the query is
            : '
        # print in_relation_string

        query = """
            PREFIX dbpedia-owl:<http://dbpedia.org/ontology
                />
            PREFIX dbp:<http://dbpedia.org/property/>
            PREFIX dbo:<http://dbpedia.org/ontology/>
            PREFIX owl:<http://www.w3.org/2002/07/owl#/>
            SELECT DISTINCT ?subject
            WHERE {

                    ?subject rdf:type ?class_type;
                             ?relation ?object.
                    FILTER( ?class_type IN (""" + in_string
                        + """))
                    FILTER( ?relation IN (""" +
                        in_relation_string + """))
            }
            LIMIT """ + str(instance_number) + """
            """

        return query
```

Get all instances related to a given instance through a set of given relationships

```
def dbpedia_get_relation_records(self, instance_list,
    relation_list, record_number):

        # 1. Build the instance list for the IN clause
        in_instance_string = ''
```

```
        for instance_url in instance_list:
            in_instance_string += '<' + str(instance_url) +
                '>,'

        # -- Remove last character
        in_instance_string = in_instance_string[:len(
            in_instance_string) - 1]

        # --debug
        # print '\n\nThe instance url list for the query is
            : '
        # print in_instance_string

        # 2. Build the relation list for the IN clause
        in_relation_string = ''

        for relation_url in relation_list:
            in_relation_string += '<' + str(relation_url) +
                '>,'

        # -- Remove last character
        in_relation_string = in_relation_string[:len(
            in_relation_string) - 1]

        # --debug
        # print '\n\nThe relation url list for the query is
            : '
        # print in_relation_string

        query = """
            PREFIX dbpedia-owl:<http://dbpedia.org/ontology
                />
            PREFIX dbp:<http://dbpedia.org/property/>
            PREFIX dbo:<http://dbpedia.org/ontology/>
            PREFIX owl:<http://www.w3.org/2002/07/owl#/>
            SELECT DISTINCT ?subject ?relation ?object
            WHERE {

                    ?subject ?relation ?object.
                    FILTER( ?subject IN (""" +
                        in_instance_string + """))
                    FILTER( ?relation IN (""" +
                        in_relation_string + """))
            }
```

```
            LIMIT """ + str(record_number) + """
            """

        return query
```