Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno Azevedo Alves da Cunha

Ulisses NextGen

November 2022

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno Azevedo Alves da Cunha

Ulisses NextGen

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
José Carlos Leite Ramalho

November 2022

STATEMENT OF INTEGRITY

ACKNOWLEDGEMENTS

I would like to acknowledge and show my warmest thanks to, firstly, my parents and my family, for their unconditional support, without them this journey wouldn't have started, let alone finished.

To my friends in Viana for keeping my spirits high and making sure it wasn't all about work, and to my friends in Braga who helped me whenever I struggled and showed me i wasn't alone

To my supervisor, José Carlos Ramalho, for his guidance and expertise. Without his input the project would have never left the drawing board.

To the folks at Untile, for giving me the space and time to work on this dissertation.

And lastly to everyone else who pushed me, even one inch closer, to completing this journey, it was truly a pleasure having you all accompany me along.

ABSTRACT

Nowadays data can have many different shapes and relations between itself, ontologies try to formalize the semantics subjacent to this data and make it understandable by humans and code alike.

While code succeeds at parsing and interpreting this formalization traditional ontology formats can be tough for a human to understand without previously deepened knowledge of the ontologic paradigm and, even then, directly analyzing a format like RDF would be, at the very least, very tedious. This problem is not exclusive to ontologic data either as to make sense of big datasets, even in famously human readable formats like JSON, humans need visualizations and abstractions.

This dissertation is a study on graph visualization of ontologic data and how abstractions can be used to convey information to the end user in meaningful ways

The information gathered is then used to implement an application called "Ulisses NextGen" that can generate an easily navigable graph visualizing application with a strong focus to support ontological data but general enough to support any information that can be abstracted as a graph. The application is served as a javascript package to be used in anywhere on the web where it can be used best to reach the end user.

Keywords: Ontology; Graph; Data; Visualizations

RESUMO

Hoje em dia os dados podem ter muitas formas e relações diferentes entre si, as ontologias tentam formalizar a semântica subjacente a estes dados e torná-los compreensíveis tanto para o ser humano como para o código.

Embora o código consiga análisar e interpretar facilmente esta formalização, os formatos tradicionais de ontologias podem ser difíceis de entender para um humano sem um conhecimento previamente aprofundado do paradigma ontológico e, mesmo assim, analisar directamente um formato como o RDF seria, no mínimo, muito tedioso. Este problema não é exclusivo dos dados ontológicos, existe tradicionalmente uma grande dificulade por parte do ser humano em interpretar grandes conjuntos de dados precisando de visualizações e abstracções.

Esta dissertação é um estudo sobre a visualização gráfica de dados ontológicos e como as abstracções podem ser usadas para transmitir informação ao utilizador final de formas significativas

A informação recolhida é então usada para implementar uma aplicação chamada "Ulisses NextGen" que gera um grafo facilmente navegável com um grande foco para suportar dados ontológicos mas geral o suficiente para suportar qualquer informação que possa ser abstraída como um grafo. A aplicação é servida como um pacote javascript para ser usado em qualquer lugar na web onde possa ser melhor utilizada para chegar ao utilizador final.

Palavras-chave: Ontologia; Grafo; Dados; Visualizações

CONTENTS

LIST OF FIGURES

ACRONYMS

A

API	Application Programming Interface.

C

CSS	Cascading Stylesheets.

D

D3	Data-driven documents.
DOM	Document Object Model.

H

HTML	Hyper Text Markup Language.

I

ISO	International Organization for Standardization.

J

JSON	Javascript Object Notation.

N

NPM	Node Package Manager.

O

OWL	Ontology Web Language.

R

RDF	Resource Description Framework.

T

TTL Turtle.

U

UML Unified Modeling Language.

V

VOWL Visual Notation for OWL Ontologies.

W

W3C World Wide Web Consortium.

1 INTRODUCTION

Ulisses was an ontology navigator developed in 2004 at the University of Minho for displaying knowledge defined in the topic maps *ISO* standard, it was developed as a way of displaying and analyzing what could be massive information data sets Librelotto et al. (2004). Since 2004, however, the landscape for concept formalization technologies has evolved beyond Topic maps to technologies and standards like *RDF* and *OWL* Horrocks et al. (2003) .

Taking into account this paradigm shift, the idea for UlissesNextGen is to be a graph data visualizer, not only for the current standards but to offer a solution that can be applied to different graph-structured information paradigms. Having this generalization in mind, this project will be developed with *RDF* and *OWL* as its main focus and directly support them as an input format.

This dissertation is intended to serve not only as an analysis of the currently available solutions for graph visualization and a description of the relevant technologies to the project but also as documentation for the development process and usage of UlissesNextGen.

1.1 MOTIVATION

With the very large amount of data available today, there exists a necessity to catalog and assign meaning to this information so that it can be understood and utilized by both people and machines in an organized and consistent way.

Ontologies show up in this context as a way of giving meaning and structure to data, making it much easier to work with. However, when applying an ontology in this kind of problem, sometimes, there arises a need to display the semantics of the ontology to the user without requiring intimate knowledge of a formal language like Turtle.

To solve this problem, ontology visualization methods are used to summarize their most important characteristics and to present them to the user in an organized way, as well as allowing smooth navigation on the semantics of the ontology and the individuals present in it. Given the usefulness of these techniques, there exist already some solutions on the market that look to tackle this issue, however, they generally suffer from scalability and/or modularity problems.

Given the large number of topics and use cases that could benefit from the use of ontologies, ontology visualization methods should provide a degree of customization to better describe each different ontology. Having this in mind, this project will be developed with modularity and customization as its main focus so that it produces a tool that can be applied in many different contexts

1.2 OBJECTIVES

The objective of this dissertation is to generate an ontology visualization and navigation tool capable of communicating efficiently and intuitively to the user the inherent semantics of the ontology being explored, as well as some of its metadata. The information must be easily navigated regardless of the size or complexity of the ontology in question. While ontologies are the main focus of the project the navigator will be developed in a way that it could realistically be used to display any kind of graph-structured data.

This tool will be implemented as a component that can be included in different projects, given that, it must be developed with several parameterization options capable of adapting to ontologies with different characteristics, thus passing its most relevant points to the consumer, regardless of the project specifications in which it is included.

In short, the expected result of this dissertation is a case study on ontological information visualization options, in order to implement an adaptable and complete navigation module that is easily included in any project that makes use of graph-structured data, with special attention to displaying ontologic data.

1.3 PROPOSED APPROACH TO THESIS

The objectives outlined in the previous approach will be accomplished by following the following list of steps in order, with some slight overlap between them.

- Bibliographic research of the state of the art
- State of the art investigating and testing
- Define a development plan and prototype based on research
- Develop the project.
- Test the developed solution across different use cases
- Discuss results

1.4 DOCUMENT STRUCTURE

This document will take a look at all the development that went into this project. Chapter 1 gave an overview of what the project is trying to accomplish and the motivation behind it.

Chapter 2 will serve mostly as an overview of of the state of the art, taking a look at the most advanced and popular techniques relating to various relevant areas to this project.

A very brief look will be taken at the proposed development approach in Chapter 3.

Chapter 4 will describe the bulk of the development endeavor of the project.

A small benchmark of the application can be found in Chapter 5 to illustrate its capabilities and limits.

Finally, in Chapter 6 an overview of the results of the project will be given as well as a small look into what could be future work to explore.

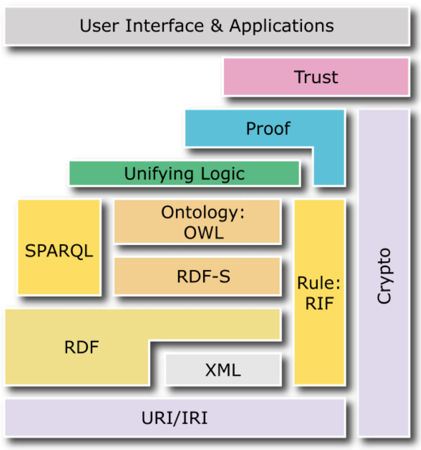
2

STATE OF THE ART

Ontologies, in the field of computer science, act as a specification of a conceptualization. Meaning that they are a formalization of a concept, they were first developed in the field of artificial intelligence. This specification of a conceptualization could be achieved in various ways, but most commonly a semantic network is used as a basis for the characterization of the knowledge at hand Gruber (1995). Since this is still a somewhat general aproach to the problem, there are various formats one could use to describe an ontology. Despite this, with the inception of the semantic web, where ontologies are a cornerstone, the W3C has standardized many of the technologies relating to it. Berners-Lee et al. (1998)

The semantic web is an extension of the tradicional web where the information being shared is readable by both humans and machines. It is supported by a list of technology standards known as the semantic web tech stack.

Figure 1: The semantic web tech stack by W3C



As can be derived from Figure 1 the W3C made *ontology web language* or OWL the de facto standard for describing ontologies. OWL is built on top of RDF and is today the most common way developers describe ontologies, as such this project will focus on supporting it and the turtle RDF sintax.

2.1 ONTOLOGY VISUALIZATION

Opposed to the technologies discussed up to this point, there is no defined standard for visualization, there are however many different projects that aim to tackle this problem.

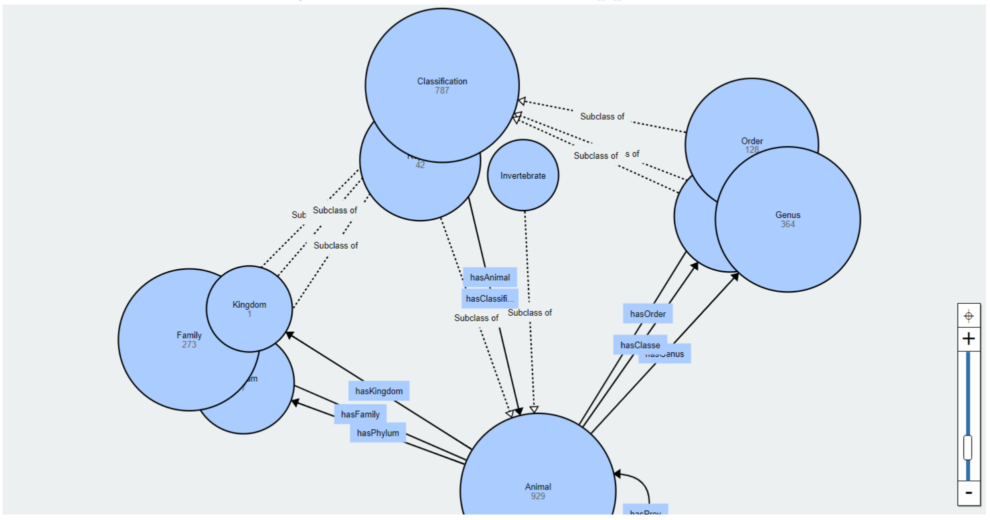
The challenge of displaying ontologies usually comes from limiting the scope of the information shown as the ontology scales as well as selecting what are the relevant parts of that information to show the user based on the characteristics of the ontology. This means that there is no one-size-fits-all that will get the best results for every case. It could also be argued that this difficulty extends beyond ontologies to nearly every kind of graph visualization so any application that tries to implement such a feature has to concern itself with tackling that issue. Schulz and Schumann (2006)

Despite these challenges, there are a few technologies that try to form a standard that can be used to display ontologic information completely and cleanly, this section will take a look at some of these formats and tools and their most important features. Dudáš et al. (2018)

2.1.1 *VOWL and WebVOWL*

This standard was defined alongside WebVOWL, a web application that implements the *VOWL* directives and also provides a *JSON* schema to encode ontologic information. An *OWL* ontology can be loaded on the web application where it will be converted to the provided *JSON* format and then displayed on to the graph, the converter to this format is available separately from the web application as a java-archive. Lohmann et al. (2016)

Figure 2: The WebVOWL Application



2.1.2 *Protégé plugins*

Protégé is a desktop application developed at Stanford University, it was developed as a tool for knowledge-based systems and is nowadays one of the most popular technologies to build and extend ontologies Musen (2015). One of the reasons for this popularity is that in recent years Protégé has enabled its users to extend the program through the use of plugins, greatly increasing its functionalities through the community, including various options for visualizing ontologies Gennari et al. (2003).

Two of the most popular plugins for this purpose are OntoGraf and ProtégéVOWL, it is also worth mentioning OWL2UML. Since ProtégéVOWL is only an implementation of the already described *VOWL* format in Protégé, this section will focus only on Ontograf and OWL2UML.

OntoGraf is a visualization tool that is shipped by default with Protégé, it is very complete, allowing for visualization of both the ontology schema as well as the individuals included in it. It also includes a collapsing mechanism and various preset layouts that serve to highlight different aspects of the ontology and guaranteeing that even larger ontologies can be correctly displayed, the main limiting factor of this plugin is the fact that it is limited to be used inside Protégé and can't be directly included in any other projects.

OWL2UML is a plugin that generates a *UML* graph from ontologic data. As expected from the *UML* format this tool is great at displaying ontologic information to the user when looking at smaller isolated areas of the graph and showing an individual node's information, however, it struggles when trying to navigate the graph because of its long, non-straight edges. Also, *UML* quickly becomes unreadable for larger ontologies.

2.2 DATA VISUALIZATION ON THE WEB

Pretty much since the inception of the web as we know it today, there has existed a necessity to display data to the user in the most varied ways. Today this task is usually accomplished by using Javascript frameworks to abstract the usually very complex manipulation of the *DOM* as well as to provide a basis for many common operations in transforming and displaying data.

The javascript ecosystem is largely powered by *NPM*, or *node package manager*. This package manager is what allows javascript applications running in the node environment (or bundled in node to run on the web) to make use of many different frameworks and utilities to power their own implementations. Having access to this kinds of tools is completely indispensable when developing larger applications in the language.

Even with the rise of virtual *DOM* frameworks like React and Vue, the de facto standard for generating data visualizations like graphs is without a doubt "*Data-Driven Documents*" or

D3. *D3* is a massive library composed of functions used to manipulate the *DOM* based on input data, as well as plenty of auxiliary methods to transform that data. *D3*'s abstractions work fairly close to the *DOM* and so the developer has full control over what is rendered to the screen. Bostock et al. (2011)

Comparing this somewhat "low-level" approach to generating visualizations to something like Chart.js which has eight, somewhat customizable, preset visualizations, we can see that *D3*'s code can be a lot more complex, but also allows for much more granular control of the rendered data. This is, clearly, a trade-off between abstraction and control.

For smaller use cases, tools like Chart.js provide a quick and easy way to generate pleasing and common data visualizations for the web, however, in the case of this project, given how much it is expected that a regular graph will be extended to incorporate ontological semantics, *D3* seems like the most logical choice for generating a custom-tailored graph visualization that can be altered to fit the display of different ontologies.

2.3 SUMMARY

When it comes to the most defining technologies and concepts in the subjects researched in this section they all share the fact that they were created to simplify the life of the end user, which is a philosophy that in many ways will guide this thesis. In that regard i'd like to highlight *VOWL* and *D3* since, in their respective field, they are able to create meaningful abstractions that can be employed or at least studied to materialize even more useful ideas and tools. While this whole section will obviously have a direct impact on the development, these two technologies will be taken in great consideration moving forward.

3

PROPOSED APPROACH

UlissesNextGen aims to be a general-purpose graph visualizer with a strong focus on supporting ontologic data. To accomplish this task it needs to propose an input format as well as a visualization standard, that will work for traditional graphs but can be extended to accommodate ontologic semantics.

3.1 TECHNOLOGIES

As discussed before the development of a complex graph visualization is best accomplished by *D3*, it natively provides directives to generate a force-directed network and doesn't have intrinsic ontological constraints while giving the developer a lot of freedom in extending it. Using *D3* means that the project will be integrated in *HTML* and will be styled with *CSS*, this has the added benefit that the program will be easily integrated across the web.

When it comes to the technologies that will be supported out of the box, the project will focus on accepting *OWL* ontologies written in the turtle *RDF* specification while also providing a traditional Nodes/Links graph input format.

3.2 INPUT FORMAT

The definition of the input format is crucial to the development and is thus a bug focus of development. Even though *OWL* ontologies will be directly supported as input, under the hood, the actual input format to the graph will be a Nodes/Links format extended to accommodate ontological information, and ontologies will be merely converted to this format.

This native format draws some inspiration from the *VOWL* standard, however, it strips it from its mandatory ontological constraints, leaving only optional and general fields to be able to provide a more customizable graph rendering from the data. For example, the *VOWL* format stores the *OWL* type of an ontologic node and decides the color it should be rendered in based on this "type" field, in the case of Ulisses however, the color is not decided by the type of a node but rather by a "color" field, when translating from an ontology to the Ulisses format we simply need to set this "color" field to whatever color we want to associate with that type. This way the graph visualizer does not need to know about what an ontologic type is, just what color to render the node in.

3.3 GRAPH APPLICATION

The graph visualizer is the biggest facet of this project and thus is where most of the development time will be spent. The main requirements for this application are as follows:

- Navigable: The graph generated need to be easily navigable by any user.
- Complete: The graph should include many versatile features so that any type of data can be appropriately rendered.
- Performant: While limited by the context of a browser the application should run without a hitch for even moderately sized ontologies.

With these requirements in mind the development will leverage *D3* and other more common web technologies to produce a result that can cater to the complete specification.

3.4 GRAPH APPLICATION

The graph visualizer is the biggest facet of this project and thus is where most of the development time will be spent. The main requirements for this application are as follows:

- **Navigable:** The graph generated need to be easily navigable by any user.
- **Complete:** The graph should include many versatile features so that any type of data can be appropriately rendered.
- **Performant:** While limited by the context of a browser the application should run without a hitch for even moderately sized ontologies.

With these requirements in mind the development will leverage *D3* and other more common web technologies to produce a result that can cater to the complete specification.

3.5 HANDLING ONTOLOGIC DATA

As mentioned previously, supporting ontologic data will be a big focus for this project, however, since the visualizer needs to be ignorant of the type of data being fed to it in order to remain generalist ontologies will be supported through the use of a translator.

This translator will serve as a bridge between data and the previously discussed input format by leveraging all of its features to make semantics shine trough to the user and simplify many of the more abstract ontologic concepts.

Development for the translator will be almost completely separate from the graph visualizer application and both will be available independently, meaning this will qualify more as an add on rather than a straight up feature.

Javascript will be the choice for developing this tool to allow simple integration with the visualizer. This option also makes sense as performance will not be a great concern since the bottleneck in that regard is likely to be the graph application anyway.

3.6 PROPOSAL OVERVIEW

As a summary of this chapter, this section will describe the proposed approach in a more broad sense.

There are two main components firstly the graph visualizing app, to be developed in javascript by leveraging the *D3* framework, this graph application will also specify an input format for the graphs it accepts.

Secondly, with the completed input format, a translator from the turtle format to it will be produced, this translator will utilize the features provided by the visualizer through the input format to generate an ontology graph.

The outlined proposed approach largely serves as a guideline for the development of the project, but does not constrain it when the situation calls for a change of plans.

4 DEVELOPMENT

The development phase is the longest phase of the project, but for the most part, the planning and proposed approach were followed and executed. The project can be split into two distinct development phases, the graph application and the ontology translator with the input format of the graph serving as the bridge between them.

Given the importance of the input format specification, it was worked on first, suffering only mild extensions as the project grew to accommodate more features, nevertheless, its first iteration served as the basis upon which the rest of the program was produced.

While the graph application and ontology translator had some overlap in their development time, the visualizer was the focus at first while the translator was finished later using the full features of the completed application to visualize the ontology semantics.

4.1 DEFINITION OF THE INPUT FORMAT

As mentioned previously the input format is the basis for the rest of the project, however, the project, in turn, affected and changed the format as new features were added.

While simple, it was imperative for the format type to be available to developers in a clear manner so that anyone could provide solutions for visualizing all kinds of data in the application, as such the format was defined in typescript and used along with it in the rest of the project.

4.1.1 *Initial implementation*

It was important for the initial implementation to be clear on what was needed of the format, as well as supporting every field present in a D3 force simulation input, while it will be looked at more in-depth in subsequent chapters it is relevant in this section to mention that the input format for the simulation is given in a list of nodes and then, separately, a list of edges between those nodes.

Given the constraints defined by d3, the minimum implementation for a nodes/edges input format would be as follows:

```
type UlissesNode extends D3SimulationNodeDatum {
    id: string
}

type UlissesLink extends D3SimulationLinkDatum {
    source: string,
    target: string
}
```

Obviously, however, it would be completely impossible to display semantics using only this information and as such the initial implementation expanded this bare-bones format into the following:

```
type UlissesNode extends SimulationNodeDatum {
    color?: string;
    id: string;
    info?: {
        [x: string]: number|string|Array<string|number>|null|undefined;
    };
}

type UlissesLink extends SimulationLinkDatum<UlissesNode> {
    color?: string;
    label: id;
    source: string,
    target: string
}
```

This was the starting point for the format, it allowed for very simple customization of the visualizations through its fields. Firstly the node object was intended to be used in the following way:

- **color:** The color the node in the visualization.
- **id:** The id of the node for the d3 simulation (This field must be unique).
- **info:** This field is an object that can include any key/value pair to make sure nodes can contain information.

The link object, on the other hand, contained more limiting features, please note the missing *"info"* field which makes it so links could not be *"weighted"*, meaning, they could not contain information besides their direction and label. As such the fields of a link object were as follows:

- **color:** The color the link in the visualization.
- **label:** The label of the link.
- **source:** The id of the source node of the link.
- **target** The id of the target node of the link.

This specification, while somewhat simple and limiting, served as the basis for making a functional prototype and as a canvas upon which future features were added as the project expanded. In the following subsections, it will be discussed how both the nodes and links specification evolved, focusing on the purpose of each field added.

4.1.2 *Evolution of the link format*

The evolution of the link format was very gradual, in this section, this evolution will be explored showing how each change was meant to affect the visualization. It is worth mentioning that there also exist a few fields, to be mentioned later, which are not required for the input format but are generated from it when it is first fed to the graph application.

The relations field

The link object is used to represent the space between two nodes in one direction, please consider a link with the following structure as an example:

```
{
    source: "node_id_1",
    target: "node_id_2",
    label: "relation_1"
}
```

This structure would represent a link from node *"node_id_1"* to node *"node_id_2"* with the *"relation_1"* label.

While this is the desired output from this object, consider that there may exist a *"relation_2"* that is also between *"node_id_1"* and *"node_id_2"*, where would this relation fit? One possible solution would be to add another link to the list with the exact same source and target while changing the label, while this approach would include all the information in the input, from a visualization standpoint these two objects represent the exact same space between nodes meaning they would overlap. Given these circumstances, the source/target pair should be unique between every link provided, so how do we deal with multiple relations in the same space?

That is where the relations field comes in, replacing the label field, and represented as such:

```
{
    relations: string[]
}
```

This means that instead of a single label, links now support an array of labels to represent N relations between any two nodes.

The "info" field

As mentioned before, to start with, the graph's links could not be "weighted", meaning they contained no more information than their direction and a name. in the case of representing ontologies this didn't come across as a problem since properties (The equivalent of links) can not carry more information. However, since this application was meant for more that just visualizing ontologies there was a need to consider the option that relations could carry any amount of information.

To answer this necessity, the exact same approach used in the case of the nodes information field was applied. This information field, much like in the nodes, is shaped as such:

```
{
    [x: string]: number|string|Array<string|number>|null|undefined;
}
```

This means that the information field is an object where information is stored in key/value pairs.

This solution, however, runs into the exact same problem as the label field had, meaning there could be two relations with different information between the same two nodes, luckily, we can easily surpass this by expanding the relations field in this way:

```
{
    relations: {
        label: string;
        info: {
            [x: string]: number|string|Array<string|number>|null|undefined;
        }
    } []
}
```

Now, instead of being just a list of strings to accommodate relations with different labels this field is a list of objects that contain both a label and an information field. Using this solution, between any two nodes, there can exist many different relations with different names and details.

The color and width fields

Both color and width share the distinction of simplest field added. They are very self explanatory in the sense that they are meant only to affect the visual rendering of the links.

The color field affects the color in which the link will be rendered, it accepts any valid *CSS* color, while the width field controls the thickness of the line. Both of this fields are optional and the application should provide defaults in case they are not present.

```
{
    color?: string,
    width?: number
}
```

4.1.3 *Evolution of the node format*

In a similar fashion to links, the node format also organically evolved in order to support more features along with the graph visualizer, in this case however the changes were much more mild as we'll discuss in this sub-section. Much like in the case of links, it is worth mentioning that a few extra fields, to be discussed later, are generated in the nodes from the initial input.

The label field

The label field was introduced to store the text to be displayed on the node when visualizing it.

Previously the id of the node was displayed instead of a label, however it is not always the case that the id of the node identifies it in a human readable way, the id field also needs to be exclusive meaning that there could not exist two nodes with the same text, to answer these problems the label field was added to, if it exists, it replaces the id in these scenarios.

```
{
    label?: string,
}
```

The visible field

The visible field, as the name implies, determines weather or not the node is visible in the initial visualization, it is worth mentioning that this field can be overwritten in the app by filtering and search functions, it only guarantees that initially, with no filtering active, the node will not be rendered.

This field also affects the links related to the respective node as links are only shown in the graph if both their source and target nodes are visible.

This field is represented by a boolean value as such:

```
{
    visible?: boolean
}
```

4.1.4 *Expanding the info field*

The info field is present in both the nodes and links input shapes, as discussed before this field is comprised of an object with string keys and values that can either be typed as strings, numbers or an array of either.

From an information storing perspective this approach did the job, however, a new feature as envisioned that when this information was rendered in the application the values could be clicked to link to a node in the graph (The effects of this *"linking"* will be discussed in further chapters). With this new feature in mind, the possible types for the values of the info object were expanded from strings and numbers to also include the InfoLink type.

```
type InfoLink = {
    label: string | number,
    linkId: string
}
```

This type functions having the label field store the text to be used when displaying the information and the linkId field storing the id of the node to be linked to when cliking the label. This feature will be discussed in detail in the following chapters but in respect to the information field of the input format, it ends up like so:

```
{
    info?: {
        [x: string]: number|string|InfoLink|Array<string|number|InfoLink>|null|
    undefined;
    };
}
```

4.1.5 *The finalized input format*

To summarize this section we will discuss the complete input format going over links and nodes and very briefly mentioning the typing of every field in the formats.

The full input format is an object storing both an array of links and an array of nodes.

```
type UlissesInput = {
    nodes: UlissesNode[],
    links: UlissesLink[]
}
```

Starting with the nodes format, it's final configuration is as follows:

```
type UlissesNode extends SimulationNodeDatum {
    color?: string;
    id: string;
    label?: string;
    info?: UlissesInfoType;
    visible: boolean;
}
```

While the link format looks like so:

```
type UlissesLink extends SimulationNodeDatum {
    color?: string;
    relations: {
        [id: string]: {
            label: string;
            info?: UlissesInfoType;
        };
    };
    source: string;
    target: string;
}
```

To finish up, the information field in both nodes and links is typed as such:

```
    type InfoLink = {
        label: string | number,
        linkId: string
    }

    type UlissesInfoType = {
        [x: string]: number|string|InfoLink|Array<string|number|InfoLink>|null|
    undefined;
    };
```

4.2 THE GRAPH APPLICATION

The graph application is the actual engine that consumes the input format previously described and generates a complete graphical visualization from it. This chapter will discuss in detail the usage of the visualizer, how the input format is used and how each of the features were implemented, going over both application architecture and technical implementation. Firstly the necessary set up to begin the app will be touched on followed by a description of the evolution from that set up.

As the program was built there were also a few implementations made to ease development and expansion of it in the future, these will be discussed in the end as a way to further improve the application should one desire to do so.

4.2.1 *Initial setup*

In order to start development on *UlissesNextGen* the first step was to initialize a *NPM* package to house the app. This step is crucial as it not only allows us to use all of the npm ecosystem it also provides liberty in customising run and deploy scripts with the *package.json* file.

Typescript

After having initialized the package the next step was to install typescript, as discussed before, given the importance of the input format specification, this was a requirement.

Typescript is superset of javascript that transpiles directly to it Bierman et al. (2014). It allows the developer to specify the types to be used in the project, which is the main appeal of the technology in this case, but it also enforces those types at transpile time, making sure that bugs don't slip through. When instaled Typescript comes with or the Typescript Compiler that can be used to perform the aforementioned transpilation according to a configuration provided in a *JSON* file.

Build scripts

After having the package initialized and Typescript correctly installed we need to set up a script that will build the application converting it to javascrit to be used in browsers. as such the followig configuration was added to the *package.json* file:

```
"build": "npx tcs -b"
```

When this build script is ran the Typescript compiler takes the typescript code in the source directory of the project and generates clean javacript code in the lib directory which functions as the entry point for the package when imported by other *NPM* apps, making sure it can be used in the browser.

4.2.2 *The application architecture*

The application was developed with ease of use in mind, as such it was designed to be used with only a target div and the calling of a function passed the graph input as an argument along with some optional configuration.

With this in goal in mind the *UlissesNextGen* package exposes only a single function that initializes the graph and returns a graph object with very limited operations beyond that.

The drawGraph function

Continuing with the previous point, the functionality of this function will be discussed right here, on a very surface level.

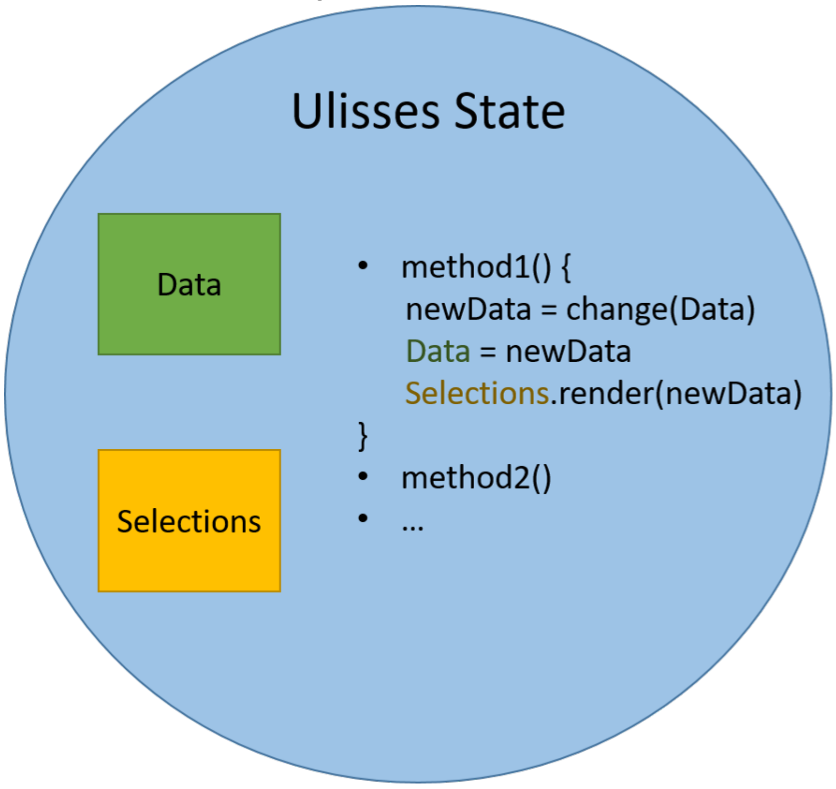
When called the function will first store the div configured by the user to receive the graph and adapt to it's size, after that it will take the input provided and generate the Uilisses application state which will store the not only the state of the data in the app but also the state of the *DOM* used for rendering it as well as all the logic linking the two.

To finish up, the function will simply return a wrapper to UlisseState with a clean up functionality to stop the execution of the graph.

The Ulisses State

The previously mentioned UlissesState can be summarized without much detail by this image:

Figure 3: Ulisses State



As represented above the ulisses state holds the state for the data in the app, this includes the following fields:

- **Selections**: This field stores the parts of the *DOM* that are used in the app as well as methods to manipulate them. It efectively renders the application.
- **Graph**: The graph field store the input information provided to the application, in the previously discussed input format.
- **Visible Graph**: Similarly to the previous entry this stores information in the input format, it is the subset of the graph currently being rendered to the screen.
- **Simulation**: This field includes the information for the force simulation that decides the position the nodes should be rendered in.

- **Filtering**: Filtering deals with, as the name implies, how to filter the input and decide what nodes and links get rendered or not. It includes fields such as graph depth, and links to consider.
- **Current Node**: The current node indicates the node that is currently in focus by the application, this node is where the whole navigation in the graph stems from.
- **Center**: The current center of the graph display, important for the position to render the nodes to.

The application operates basically in the same loop stemming from this state object, once any of the data is updated, a method in the selections field is called and passed the updated data rendering it to the *DOM*.

4.2.3 *Setting up D3*

In order to implement most of the functionality on the app there will be a requirement to perform major manipulation to the *DOM*, this would be a major hurdle using the native browser *API*. D3 is a javascript library made with the purpose of rendering data in *HTML*, it also comes equipped with many utils for manipulating a positioning that same data, considering this it is the perfect tool for a project such as *Ulisses*.

After installing *D3* from *NPM*, there are 3 relevant aspects of the framework that will come into play in the development, the selections *API* along with the *Enter and Exit* functionality, the dynamic properties *API*, and the force simulation module. All of these will be discussed in detail right ahead.

Selections

Selections are a feature in d3 that allow developers to select *DOM* nodes and store them in variables to be manipulated

Given the nature of this project selections are crucial to render information to the user, this is possible because *D3* selections are actually capable of being linked to data, making sure that the *DOM* is updated according to changes to that same data, this is enabled as mentioned before, because of the *Enter and Exit* functionality. For example take a look at the following piece of code:

```
const nodes = graphSelection
    .selectAll("#node")
    .data(visibleNodes, node => node.id);

nodes.enter()
    .append('circle')
    .attr('id','node')

nodes.exit()
    .remove()
```

This is a simplified excerpt from the application's code, it deals with rendering the nodes according to the data in the variable *visibleNodes*. As you can see, we first select every *DOM* node with id *node* and link the *visibleNodes* array to that selection.

Once we have the selection with linked data we can use the *enter()* and *exit()* methods to decide what to do when data (as the name implies) enters or exits the array. s In this case if a new node appears in the array we simply append a circle *DOM* node with id *node*, and if an element is removed from the array we remove the *DOM* node associated with the missing data.

To sumarize, selections are the way the application interacts with the *DOM*, acting as a view, to power the entire visualization shown to the user, in addition with the power of the *Enter and Exit API* the application can easily and intuitively update the view according to changes in the subjacent data. In *Ulisses* this will mainly be used to manipulate the elements that represent the graph.

Dynamic properties

Dynamic properties are the way *D3* influences attributes in the *HTML*, in this application it is used mostly to add content and css styling to the app, but it also has other uses, please take a look at the code bellow.

```
const nodes = graphSelection
    .selectAll("#ulisses")

nodes.text('Node text')
nodes.style('background', 'blue')
nodes.('href', '#')
```

In the snippet shown, we can take a look at 3 ways *D3* handles dynamic properties.

Firstly all the *DOM* nodes with id *ulisses* are gathered in the *nodes* selection and then various methods can be called on it to apply these dynamic properties to every node in that same selection.

These methods can be described as follows:

- **The *text()* method:** This method is used to add actual content to the *DOM* nodes, for reference the call in the example above would result in this html:

```
<div id="ulisses"> Node text </div>
```

- **The *style()* method:** The *style* method is used as the name implies to add *CSS* styling to the node in the selection, this style is added inline to the html, This approach is great because instead of providing actual *CSS* files to style the app it can all be done with just javascript. Taking a look at the example of this method above, it would produce the following *HTML*.

```
<div id="ulisses" style="background: blue;" />
```

- **The *attr()* method:** This method can be utilized to set *DOM* node attributes besides style. It is useful for setting attributes like *href* and the properties of various svg elements that can only be styled that way.

```
<div id="ulisses" href="#" />
```

Force simulation

Now with the knowledge that *D3* can handle the rendering of information to the screen how can we correctly decide the position of the nodes so that the graph is correctly rendered? Luckily *D3* also provides us with the *force simulation* module, this module is capable of generating a simulation of a force directed graph.

In the simplest terms this feature takes the graph input format as input and maps the nodes with position coordinates of where they shoukd be rendered in the graph according to a force directed simulation. This simulation can support various forces but we'll take a look at the ones actually used in Ulisses.

Bellow is included the actual code for the generating the Ulisses simulation

```
    ulissesState.simulation = d3.forceSimulation(ulissesState.visibleNodes)
        .force("link", d3.forceLink()
            .id((node: UlissesNode) => node.id)
            .links(ulissesState.visibleLinks)
            .distance(250)
            .strength(0.2)
        )
        .force("charge", d3.forceManyBody().strength((d: UlissesNode) => {
            if (d.id == ulissesState._currentNode.id)
                return -10000
            return -5000
        }))
        .force("x", d3.forceX(width / 2).strength(0.3))
        .force("y", d3.forceY(height / 2).strength(0.3))
```

As you can see, there are 4 forces that govern the graph in the application, they will all be explained in detail in this section.

- **The *link* force:** This force has the job of making sure that nodes connected together by links get strongly attracted to eachother, it is programed to take priority over other forces in the graph that could separate linked nodes.
- **The *charge* force:** The charge force gets it's name because it makes nodes behave as if they had a magnetic charge, which we can use in this case to guarantee that nodes dont clump together by giving them all a negative charge which will cause them to repel from eachother.
- **The *x* force:** This force as the name implies makes it so the nodes are attracted to a certain X coordinate in the graph. It is used in this case so that the nodes spread from the middle of the graph.
- **The *y* force:** This force behaves much like the previous one with the exception that it influences the node's y coordinate. It also has the exact same use here.

With all four of these forces we can guarantee that an easily navigable graph will be produced. where the nodes will be placed close to the center (thanks to the x/y forces) but will not overlap (because of the charge force repulsion) and nodes with links between them will clump together for easy navigation (on account of the link force).

4.2.4 *Graph navigation*

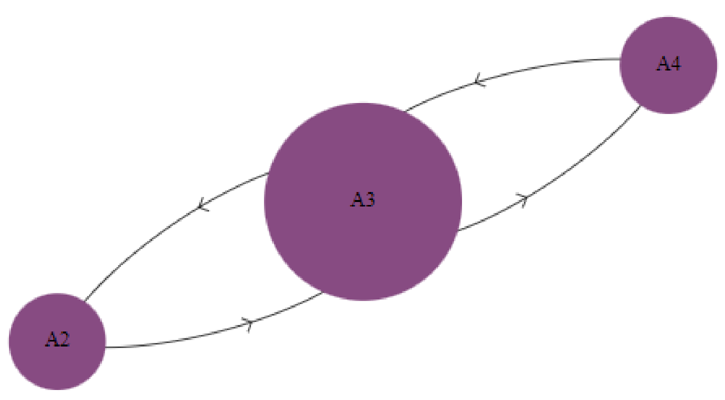
As the main proposition of the graph application, ease of navigation was a top priority when building it. In the end, the navigation method chosen relies on a centered node in the visualization from where the rest of the graph flows, this centered node can then be accessed to consult its information or another node can be selected to take it's place shifting the graph.

With the previous paragraph in consideration, the selected node is, in essence, the way ulisses handles navigation in the graph, so in this next section we'll take a look at why this was a necessity, what it entails, and how it was implemented in the app.

The selected node

The selected node is, as mentioned before, a central concept to ulisses, bellow is an image of what it looks like in the graph application.

Figure 4: The selected node



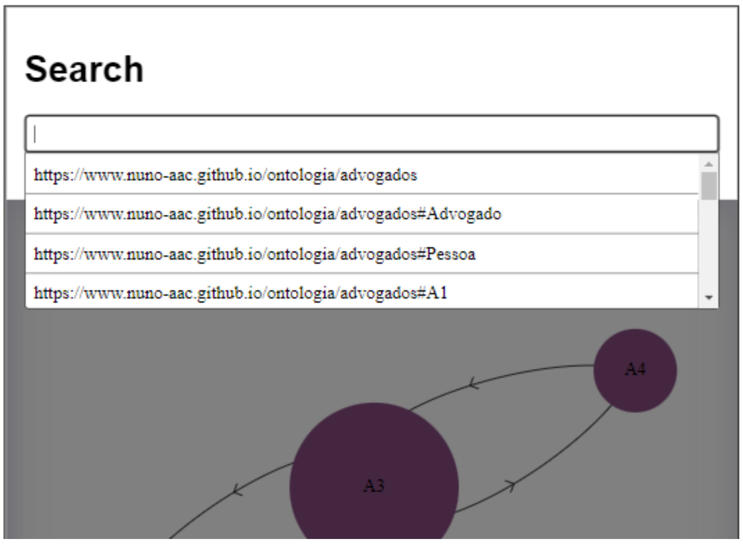
As you can see, one of the nodes is the largest and centered in the screen, that marks it as the selected node, unlike other nodes it has a fixed position, meaning it is not affected by the force simulation that moves nodes around it while still affecting them, meaning that nodes with links to the selected node will also be dragged close to the center by the simulation.

Nodes linked to the selected node being close to the center is relevant because navigation in the graph is handled by clicking any node other than the currently selected one and making it the centered node, this is made in this way because when navigating a graph you usually want to check information in relation to the node you are currently visualizing and this approach provides a very natural way of doing it. You can also note that nodes get smaller as they get farther away from the center, in a variation of the fish-eye effect, this is based on the same assumption as before that nodes that are not closely linked to the current node are not as relevant to the user.

When a node is clicked, the current selected node is unfixed from the center of the simulation (meaning the force simulation can now act on it), and the clicked node is set as the new selected node, what this means is that it is no longer affected by the simulation and will be artificially dragged to the center of the canvas (along with every node linked to it). This creates an experience where the user can easily traverse the graph just by clicking linked nodes on the screen.

While this approach was a great success in producing easily navigable graphs it overlooks one situation. This strategy was developed under the assumption that nodes closer in the graph to the selected one are more likely to be navigated to by the user but this may not always be the case, sometimes the user may elect to jump to a completely unrelated node in the screen. While there is no constraint in the graph that forbids the use from clicking a node farther away from the center this is not ideal, especially because the user may have a specific node in mind that would be hard to find just by searching the small nodes far away from the center. In order to solve this problem, the following feature was added to the graph:

Figure 5: The search drawer



https://www.nuno-aac.github.io/ontologia/advogados
https://www.nuno-aac.github.io/ontologia/advogados#Advogado
https://www.nuno-aac.github.io/ontologia/advogados#Pessoa
https://www.nuno-aac.github.io/ontologia/advogados#A1

This small drawer in the top of the graph provides the user with a search bar that can be used to search for any node in the graph, when any of the options is clicked the drawer is closed and the node associated with that option is set as the selected node. This gives the user an alternative to the local navigation in the graph

Node information

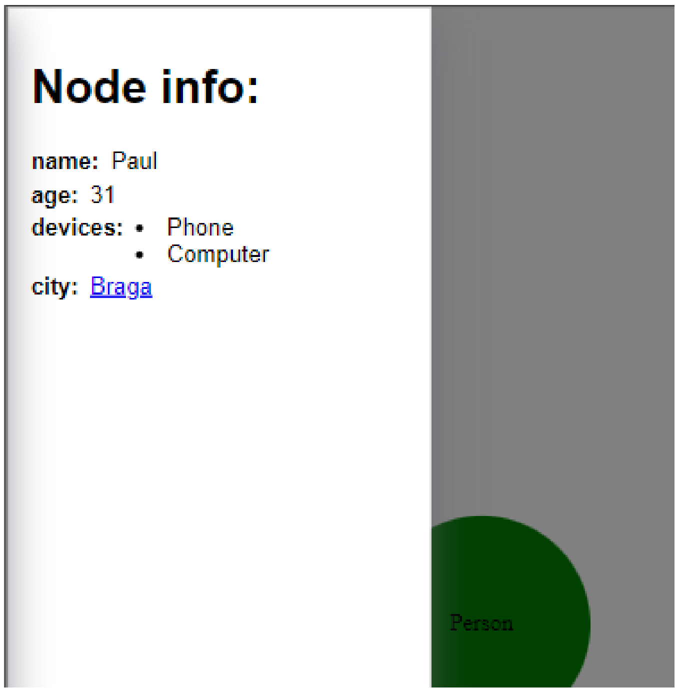
While the selected tool is a great asset for navigating the graph we discussed before that nodes could be weighted, as a refresher here is the information field on the node input format:

```
info?: {
    [x: string]: InfoLink|number|string|Array<string|number|InfoLink>|null|
undefined;
};
```

As you can see there is information stored in the nodes in the form of an object with key value pairs. Where these values can be typed as a number, a string, an InfoLink (which is just a number or string with a different node id associated) or an array of any of those.

So where is this information object in the graph? As it turns out the selected node has a part to play in this matter as well. While clicking any other node in the graph causes it to shift, if the centered node is clicked it actually opens a drawer in the application that renders this information.

Figure 6: Node information drawer



Node info:

name: Paul
age: 31
devices:
- Phone
- Computer

city: Braga

Represented in the example above are the 4 types of data the drawer can render, firstly a string with the name of the person represented by the user, then a number representing the age, an array for the devices belonging to the person and a *LinkNode* that not only presents the person's city, it will make it so when the link is clicked the drawer will close and the selected node will be replaced by city of Braga's node.

Here we can also analyze the true value of the *LinkNode* type, imagine a graph where one of the nodes is linked to almost every other (like every person lives in Braga), instead of polluting the graph visualization with many links that all have the same meaning you can instead opt to store that connection in each individual nodes information.

Rendering nodes

Rendering the nodes to the screen is actually a very simple task since in the *DOM* the nodes are respresented with just a colored circle element and a bit of text. Bellow you can find an example of a node in the *DOM*:

```
<g id="node" transform="scale(1)" style="transform-origin: 326px 425px 0px;">
    <circle r="75" style="fill: rgb(135, 75, 130);" cx="326" cy="425" />
    <text alignment-baseline="middle" text-anchor="middle" x="326" y="425">
        Node Text
    </text>
</g>
```

As you can see a node contains only an group with two elements. This minimalist approach actually has the benefit of improving performance for ontologies with a large number of nodes.

4.2.5 *Links in ulisses*

Links play a big part in the visualization of the graph, not only are they obviously very important for navigation they also have the ability to be weighted and contain information of their own. They also present unique challenges when rendering them to the user.

With the last paragraph in mind the following section will take a look at how links are handled in Ulisses.

Preprocessing links

Unlike nodes, links have a few particularities when in comes to their rendering and require a bit of manipulation before they are ready to be fed to the application.

Firstly, links lack the visible property present in nodes that decides if they should be rendered in the graph, instead, the condition for rendering a link is actually both nodes connected to it being visible themselves, it would make no sense to have a link leading from nothing or to nothing.

Another important aspect is that links need to have awareness of other links in the graph while nodes can be completely self contained. To understand why this is the case please consider the following 2 nodes and links

```
const node1 = {
    id: 1
}

const node2 = {
    id: 2
}

const link12 = {
    source: 1
    target: 2
}

const link21 = {
    source: 2
    target: 1
}
```

In this case, the two nodes have links between them going in opposite directions, however if the two links ignore eachother's existence there will be two overlapping lines between the nodes, which does not represent to the user the two links that exist in reality.

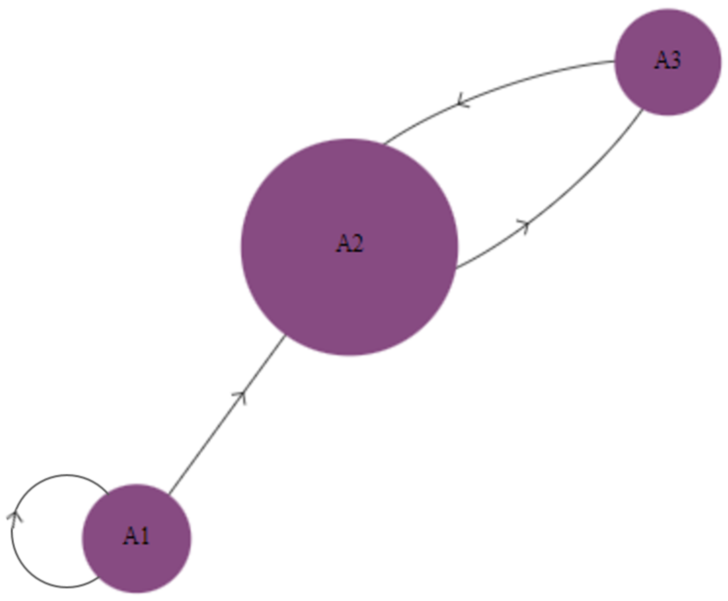
Because of this scenario, when first loading the links to the application they are preprocessed to determine if they have an opposite peer and, if they do, we add a *hasOpposite* field with a value of true to the link. This adds some negligible overhead to the graph loading, however the value of this field can also be set beforehand in the input and the application configured to ignore this step.

Rendering links

With the preprocessing out of the way and the input passed on to the visualization the main challenge is rendering the links on screen. While nodes are basically circles with a few added details, links actually have a lot more complexity to them.

When it comes to information for rendering the links there are really only two points to go off from and they are the position of the respective start and target nodes, now if all that was needed was a line between these two points the task would be completed but this is not the case, bellow we can find an example of the types of links actually rendered in the graph.

Figure 7: Link Types



As you can see not only do the links need to represent direction they also need to curve if they have another link opposite to them, and curve differently if their respective nodes have different sizes. So how can we achieve this from the the information available to us? The answer is a bit of geometry, without getting into actual mathematics bellow you can follow the approach taken for each problem

- **Rendering direction**

  In order to represent direction in the links we can observe small chevrons pointing in the desired direction.

  To achieve this effect first we had to find the middle point between the two nodes, after that we generate a vector with the size desired for the arrow and rotate it to find the position for the two tips of the chevron. After having the position of these tips we just have to draw a line from the middle of the link to the generated points.

- **Curving the links**

  As mentioned before when there are two links with opposite direction there exists a need for them not to overlap, in addition to that, since the nodes change sizes in order to highlight the ones closer to the center with the fisheye effect, the links also have to accommodate that difference in size between them.

  To tackle these challenges ulisses uses the power of bezier curves, in the case the nodes match in size the vectors used for the curve will be equal size but rotated in opposite ways, on the other hand, if one of the nodes is bigger, the vector from that side will also grow and rotate more compared to the smaller side.

  Besides the size difference, the angle of the vector will also be affected by the distance of the nodes. Here is a graphical example of all these techniques:

  In case of self referencing links, whose source and target are the same node a shifted circle will instead be used to represent the curve

Link information

When discussing the input format in the previous section it was determined that the link format was not meant to directly represent relations between the nodes but rather the space between them in the graph. As a reminder here is the *UlissesLink* format.

```
type UlissesLink extends SimulationNodeDatum {
    color?: string;
    relations: {
        [id: string]: {
            label: string;
            info?: UlissesInfoType;
        };
    };
    source: string;
    target: string;
}
```

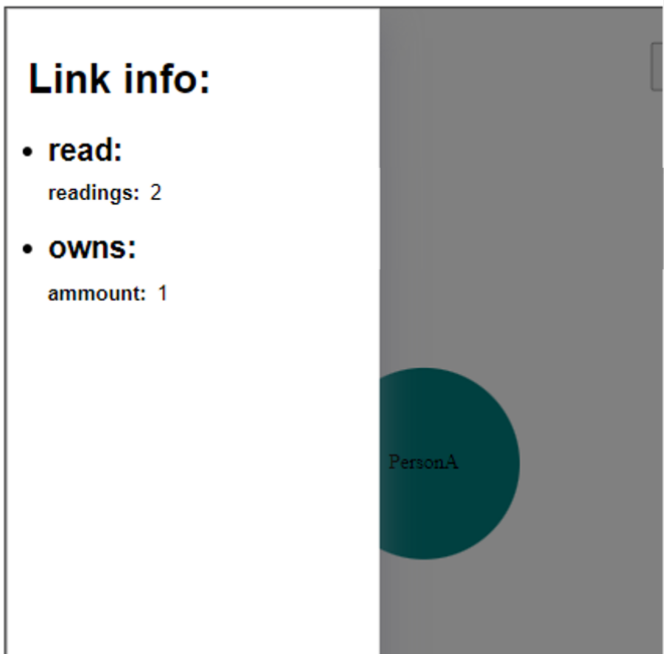
As you can see the actual information for the relations between two nodes is actually stored in the relations (to differentiate them from links) field. This field is actually an array that can contain multiple different relations, for example:

```
{
    source: "Person1",
    target: "Book",
    relations: [
        {
            label: "Owns",
            info: {
                ammount: 1
            }
        },
        {
            label: "Read",
            info: {
                readings: 2
            }
        },
    ]
}
```

In these example is represented a link between two nodes **Person1** and **Book**, this link includes two relations, **Owns** and **Read**, these relations are also weighted to show that the book was read 2 times but the person only owns 1 copy. But where is this information represented in the graph application?

Much like the nodes clicking a link in the graph causes a drawer to open, using the previous example as a base, if the link was clicked the following would be presented to the user:

Figure 8: Link information drawer



Link info:

- read:
  readings: 2
- owns:
  ammount: 1

As you can see this drawer functions very similarly to the previously discussed node drawer, except it can render multiple info objects if more than one relation in the clicked link includes it.

4.2.6 *Filtering and depth limiting*

As discussed before the graph application runs completely the client side, initially this raised some performance concerns, especially about rendering larger graphs, this will be looked at in greater detail in the benchmarking chapter further ahead. Another problem with bigger graphs is the fact that a large number of nodes and, especially, links can make the ontology completely unreadable on the screen.

Since the large number of nodes and links presents more of a challenge to the rendering rather than preprocessing and storing the input, a solution was implemented that focuses on limiting and curing the information shown on screen to the user.

To achieve a more limited scope of information, a concept called a subgraph was used. A subgraph is a subset of the original graph's edges and nodes. This is an important concept because we can use it to avoid showing excessive information to the user on larger graphs by instead selecting only the relevant information and rendering it in place of the full graph. The next challenge is how to actually select this subset.

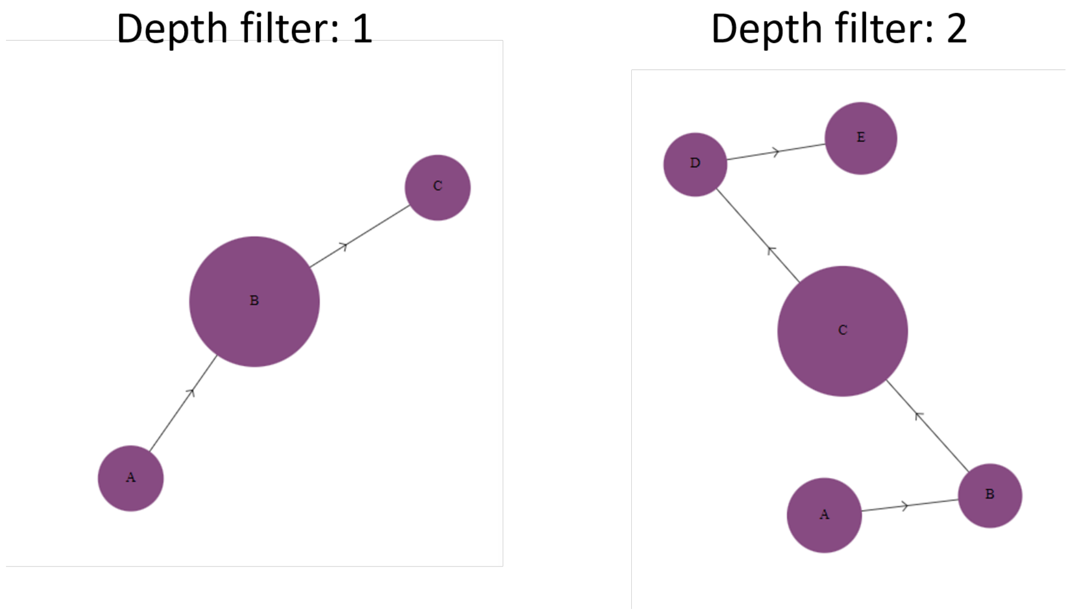
The subgraph is always generated from the currently selected node since it is assumed to be the most relevant to the user at the time, from there we can use different techniques to reduce the graph.

Depth limiting

When it comes selecting which information is more relevant it is usually safe to assume that nodes positioned farther away from the selected node or not connected to it at all, will be less relevant than those more closely connected. Depth limiting comes to the rescue as a way to show only information close to the selected node.

The depth limiting filter works by only selecting nodes that are separated by an arbitrary number of links from the selected node. Bellow you can find an example of how different values of depth limiting affect a graph.

Figure 9: Depth filter



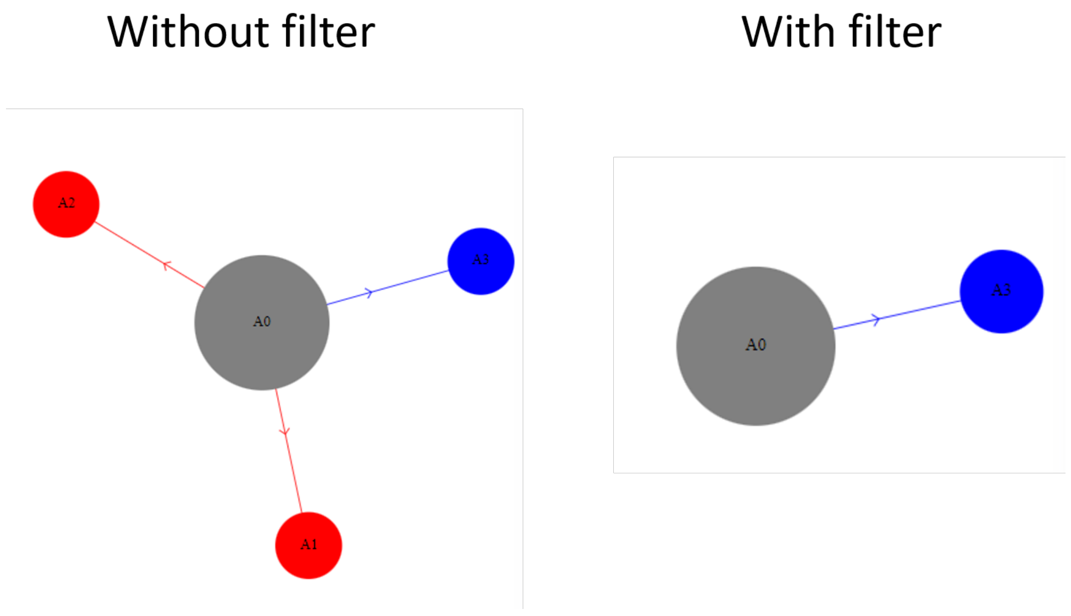
As you can see depending on the needs of the user the filter can be tailored to show more or less information

Link filtering

While depth limiting is usually really effective at limiting the size of the graph on screen it can actually be remarkably ineffective in one particular scenario, that is if a node has a very large amount of links connecting directly to it.

When dealing with a situation such as this one the graph also supports link filtering where certain links can be excluded from the filtering process. For example, the graph can be configured to ignore all but the filters with id *blue*

Figure 10: Links filter

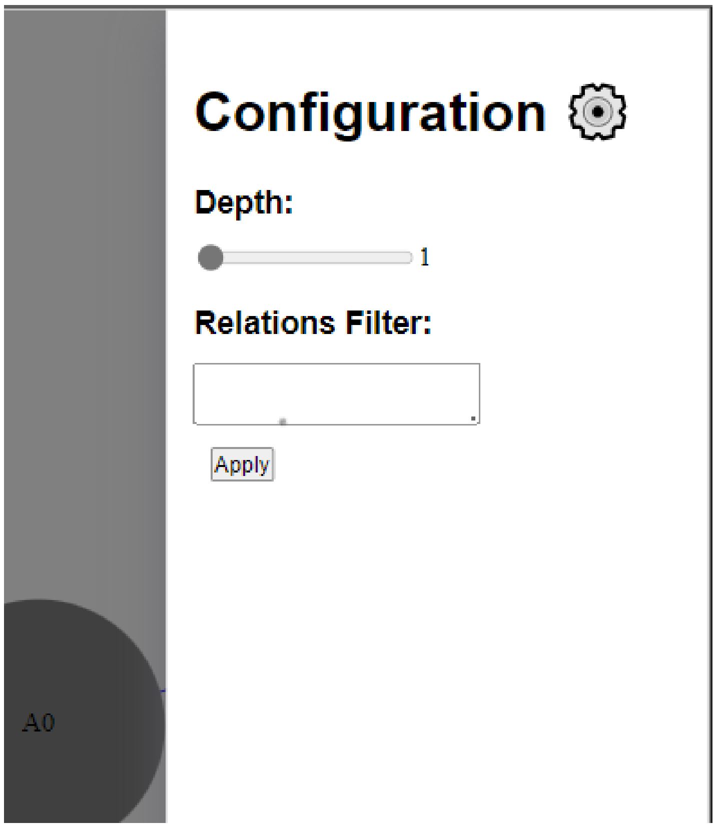


Besides the benefits in limiting the size of the graph, this filter also has great value as a semantic filter for the user to easily navigate the graph and find the required information.

4.2.7 *User selected filters*

Now that we've established what filters are in place how can the user actually apply them to their specific use case? This feature is handled by a button in the app that causes a drawer to be opened that gives the user direct control over the filtering.

Figure 11: The filters drawer



There is also an option to configure the graph with some preset filters so that the end-user does not have to manage that himself.

4.2.8 *Graph application configuration*

The *drawGraph()* function actually accepts a bit more than just the *Ulisses* input format, it can also accept a small configuration object shaped like so:

```
{
    filter?: {
        depth?: number,
        links?: string[]
    },
    defaultNode?: string
}
```

The filter field can be used to preset the graph filters as described in the previous section while the *defaultNode* describes what node should be the selected node when the graph first loads

4.3 THE TURTLE TRANSLATOR

Since *Ulisses* was made as a general purpose graph visualizer there is no direct support for accepting ontologies as an input format. However the beauty of a general purpose application is that it can be used for anything, with correct translating of the ontology to the *UlissesInput* format the visualizer is more than capable of rendering that kind of information.

The turtle translator is an add-on to *Ulisses*, it was built as a completely separate package that as the name indicates can translate from the turtle (*.ttl*) format to *UlissesInput*. In this section we'll go over how the translator makes use of the several features of the visualizer to render an ontology to the user.

It is worth noting that this translator is not an absolute truth for rendering ontologies, as discussed before, different ontologies can have wildly different configurations, and users are encouraged to leverage the *UlissesFormat* to cater to their own needs. What the provided translator means to achieve is a balanced ontology rendering method that will work well for as many ontologies as possible, but you will usually be able to do better when working with a specific example..

4.3.1 *The translateOntology() function*

Similarly to the graph visualizer package, only a single function is exported by the ontology translator, the apropriately named *translateOntology* function. This function takes in a single parameter that should be a valid ttl string.

Parsing the ontology with N3

Inside the function once we have access to the passed string we make use of one of the fastest ontology parsing packages in the ecosystem. This package is called *N3.js* and it no only validates the provided argument as a valid ttl string it also returns a stream of triples that can be parsed and translated into the *Ulisses* format. Here is a small snippet from an example ontology.

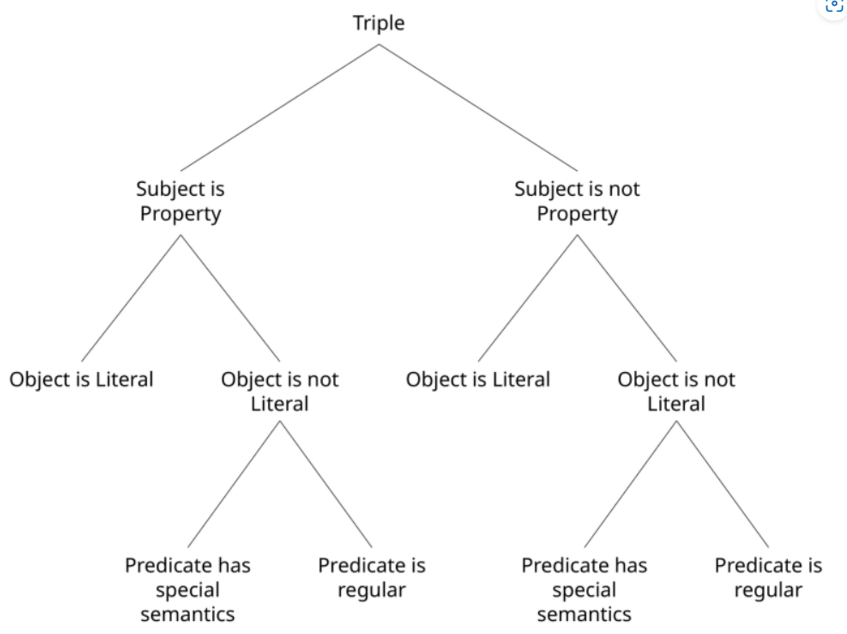
```
:PersonA :likes :PersonB;
    :height "180cm".
```

This snippet includes a *PersonA* that likes *PersonB* an is 180cm tall, and here is the output from passing it through *N3.js*.

```
[
    {
        "subject": {
            "termType": "NamedNode",
            "value": "http://www.semanticweb.org/ant niocarvalho/ontologies/
    animals#PersonA"
        },
        "predicate": {
            "termType": "NamedNode",
            "value": "http://www.semanticweb.org/ant niocarvalho/ontologies/
    animals#likes"
        },
        "object": {
            "termType": "NamedNode",
            "value": "http://www.semanticweb.org/ant niocarvalho/ontologies/
    animals#PersonB"
        }
    },
    {
        "subject": {
            "termType": "NamedNode",
            "value": "http://www.semanticweb.org/ant niocarvalho/ontologies/
    animals#PersonA"
        },
        "predicate": {
            "termType": "NamedNode",
            "value": "http://www.semanticweb.org/ant niocarvalho/ontologies/
    animals#height"
        },
        "object": {
            "termType": "Literal",
            "value": "180cm"
        }
    }
]
```

Here you can see the type of triples returned by *N3*, they contain a subject, a predicate, and an object, the translation of the ontology is performed by looping through these same triples. Each triple is then run through a decision tree to determine how to correctly translate them to the *Ulisses* input format. Here is a high level overview of this decision tree

Figure 12: The Ulisses Prototype



Every node in this tree will be analized in detail in the following sections along with the way *OWL* and *RDF* specific semantics were handled.

Is subject a property?

When a triple is received by the translator the first decision to make is whether or not the subject of the triple is typed as an *OWL* property. It can be either a *DataProperty* or an *ObjectProperty*.

It is important to make this distinction between properties and other nodes because properties need to include a domain, and possibly a range, it also make sense to represent them as links in the graph (even though they are indeed node in the turtle format), because they can be used to link individuals to other individuals or literals. If the domain is not provided it is assumed to be *owl:Thing*

If the subject is not a property it will be added as a node linked by the predicate to the object of the triple.

So to summarize we can look at an example of both these cases being converted to the *Ulisses* format:

```
:likes a owl:ObjectProperty;
    owl:Domain :Person;
```

The output of translating these object property triples would be as follows:

```
{
    nodes: [{ id: "Person" }, { id: "Thing" }],
    links: [{
        source: "Thing",
        target: "Person",
        relations: [{
            label: "Likes",
            info: {
                type: "ObjectProperty"
            }
        }]
    }]
}
```

As you can see, this object property generates 2 nodes, one for the range (assumed to be *owl:Thing*) one for the domain (specified to be *:Person* and the actual information for the object property is stored as a relation in the link between them.

We can also take a look at the opposite example where the subject of the triple is not a property.

```
:PersonA :likes PersonB;
```

In the *Ulisses* format output for these two triples we find

```
{
    nodes: [{ id: "PersonA" }, { id: "Person B" }]
    links: [{
        source: "PersonA",
        target "PersonB",
        relations: [{ id: "likes" }]
    }]
}
```

Similarly to the first example it generates two nodes and a link, however, this time the information is stored in the subject node and the link serves only as a semantic connection between the nodes, without any weight to it. Note that the link from this example is completely separate from the previous one even though they share an id in their relations, what uniquely identifies a link is the source and target fields as a compound key.

Now that we've established the importance of separating these two types of triples the next step will evaluate the object of the triple.

Is object a literal?

The object of a triple can be of two distinct types, it can be a node in case the predicate is an object property or it can be a literal in case the predicate is a datatype property. The triple is handled differently based on this type.

If the object is a literal then that object will be added to the info field of the subject. Other wise the predicate will be added as a link between the subject and object node. This can be made clear looking at an example.

```
:PersonA :likes :PersonB;
    :eats "Cake"
```

These two triples would translate to the following format:

```
{
    nodes: [{
        id: "PersonA"
        info: {
            eats: "cake"
        }
    },
    {
        id: "Person B"
    }]
    links: [{
        source: "PersonA",
        target "PersonB",
        relations: [{ id: "likes" }]
    }]
}
```

As you can see the first triple is reflected as a link between person A and B with the relation "likes", meanwhile the second triple (with a literal object) was instead added to the *PersonA* info field with a key equal to the predicate and a value equal to the object.

With this section and the previous one we can understand how triples translate into the graph format, but a list of triples is not an ontology so we'll take a look at how actual *OWL* and *RDF* semantics translate into our graph.

Is predicate from OWL/RDF?

While most triples are handled simply by the previously discussed sections sometimes *OWL*/*RDF* semantics get involved and the can be represented in a more appealing way given their importance to the semantic meaning of the triple.

A easy example to give in terms of the relevance of these semantics to the rendering of the graph comes from the *rdf:type* object property. Usually a relation like this would produce a link between two nodes however does that make sense every time?

The most numerous type of node in an *OWL* ontology are usually the named individuals, so if we accept the previously discussed technique all of them would link with a *rdf:type* link to the *owl:NamedIndividual* node, this could easily be thousands of links coming from the same node for certain ontologies, which is completly unworkable for rendering purposes. So how do we translate semantics like this to the graph?

Translating rdf:type

With the described problem in mind, the way Ulisses handles node typing is by not representing it as a link in the graph, instead this information is stored in the respective nodes info field, if the object happens to be one in the list, the color of the node will be adjusted accordingly:

- **owl:Class**: Yellow
- **owl:NamedIndividual**: Purple
- **owl:DatatypeProperty**: Green

Perhaps you noticed that *ObjectProperties* are missing from this list, this is due to the fact that *ObjectProperties* are not represented by nodes but rather only by links as discussed before.

As an example of how this relation is handled in translation we can look at the following snippet and it's output.

```
:PersonA  rdf:type  owl:NamedIndividual,
                    :Person.
```

```
{
    nodes: [{
        id: "PersonA"
        info: {
            type: ["NamedIndividual", {
                label: "Person"
                linkId: "Person"
            }]
        }
        color: "purple"
    },
    {
        id: "Person"
    }]
    links: []
}
```

As you can see the output produces no links and the type information is instead stored in the info field for the *PersonA* node. It should be noted that while the link is omitted from the input format the type (if not from owl semantics like NamedIndividual) is stored as an *InfoLink* that will allow direct navigation to the type node from the information drawer in the app. This maintains ease of navigation without sacrificing information and the readability of the graph (even improving it).

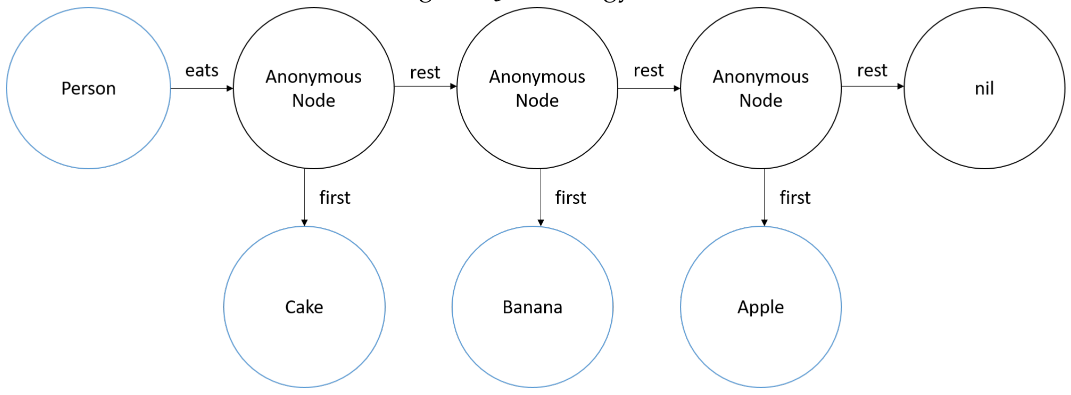
Translating lists

Lists are a very particular form of writing rdf data, when a list is used it is actually represented in triples as a linked list. This is much more obvious graphically so bellow you can find an example for the following snippet

```
:Person :eats (:Cake :Banana :Apple).
```

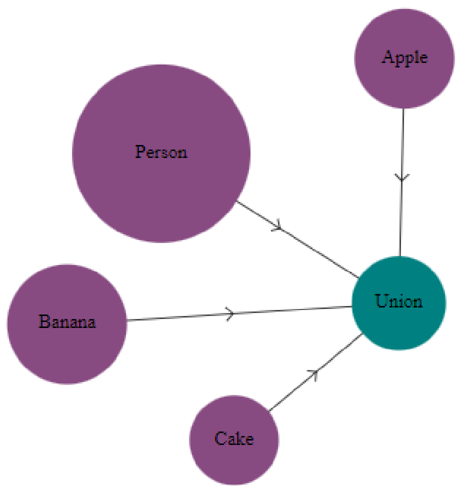
The way this snippet is translated to triples is very much as a linked list, a representation of the graph generated by the previous snippet can be found bellow:

Figure 13: Ontology list



As you can see this representation is powered by anonymous nodes and greatly hinders readability, in *Ulisses* lists are represented as groupings of nodes this loses the order of the list but for the most part lists are used to group *RDF* objects together rather that to order them. So by sacrificing a small amount of information we can produce a much more readable output.

Figure 14: Ulisses list



This example is the output from feeding the snippet above to ulisses.

Translating DatatypeProperties

While we discussed properties earlier in the document it is worth exploring here how a datatype is represented in the graph since it differs from every other type.

Before it was discussed that Properties end up represented as links instead of nodes in the graph, this is the case because *ObjectProperties* require a domain and a range, even if implicit. This is not the case for *DatatypeProperties* since they only require a domain (that once again may be implicit).

The consequence is that while the link related to the property will always have a source, a target may not be present so, instead, a node is created to fill that purpose. This node will simply replicate the information of the link.

The range while not required can be included and will stem from the *DatatypeProperty* generated node.

Here is an example of all of these features in action from the following snippet:

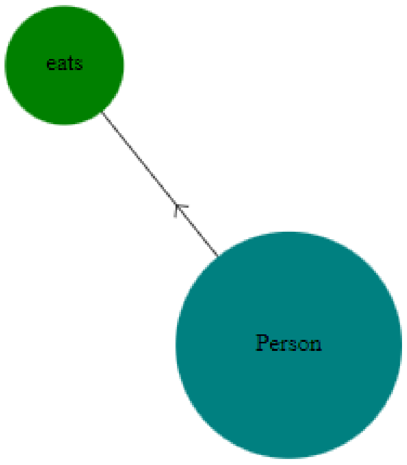
```
:eats a owl:DatatypeProperty;
    :description "What a person eats";
    owl:domain :Person;
    owl:range xsd:string.
```

This *TTL* specifies a datatype property with a domain of *Person* and a range that accepts *string*, it also provides a small description of the property. The translator would handle this input and return these nodes and links:

```
{
    nodes: [{
        id: "Person"
    },
    {
        id: "eats",
        info: {
            description: "What a person eats"
        }
        color: "green"
    },
    {
        id: "string"
    }]
    links: [{
        source: "Person",
        target: "eats",
        relation: [{
            id: "eats",
            info: {
                description: "What a person eats"
            }
    }, {
        source: "eats",
        target: "string",
        relation: [{ id: "range" }]
    }]
}
```

And this input would translate graphically to:

Figure 15: Datatype property



4.3.2 *Final considerations for the translator*

With the full overview for the translator package it is now fair to take a look at it's effectiveness in rendering ontologies, even according to some benchmarking ontologies.

The translator takes on the difficult job of not only reproducing the semantics of the ontology in graph form but also rendering its individuals, Usually, ontology visualizers and benchmarks concern themselves with only one of the sides, but here we try to somewhat unify these two aspects to cover as many ontology types as possible.

With a very generalist approach no information is lost in the converting from *TTL* to the *Ulisses* format. What this means is that both semantic ontology benchmark and individual visualizers validate the way *Ulisses* renders ontologies, however, they do so by compromising on actual specialization on either of these, this causes some cases where the user would want a focus on a specific aspect of an ontology and finds the graph polluted by all the information that a generalist solution has to provide.

With the previous paragraph in mind, this translator is very good at doing what it was designed to do which is, given and ontology it will produce a navigable, human understandable, graph in a format that can be ingested by the visualizer. It also works as a basis for developers looking to develop their own translator more fitting for their own use case. In this this respect, we can conclude the translator was a success.

5 BENCHMARKING

One of the requirements outlined at the start of development for the grpah application was that it was performant, meaning that it should be able to handle even moderately sized graphs. This section will take a look at if and how well this specification was met.

All the tests in this section were run on a single desktop device and are averaged from the 3 most popular browsers (that all yielded pretty similar results).

There are two metrics from which we can measure the performance of the application, firstly we have the rendering performance, this metric refers to the amount of information that can be displayed at once on the screen. On the other hand we can take a look at startup time for the application, since *Ulisses* needs to perform some expensive manipulation to the data at load time this can take some time. Both of these metrics will be analyzed separately.

5.1 RENDERING PERFORMANCE

As mentioned before, the rendering performance test measures how many nodes can be rendered at once on screen. When developing the app, this was one of the main concerns even trying to improve the performance by adding some overhead when loading the data. This section will take a look at if these efforts paid off.

The test was performed by gradually increasing the amount of nodes on screen and assessing the behaviour of the application, in this case the metric used to measure performance was usability.

The findings suggest that the application behaves seamlessly until about 200 nodes and links on screen. After 200, performance begins slowly deteriorating but the graph remains usable until around 500 nodes and links on screen, after that point the stuttering becomes unbearable with the graph freezing for up to 5 seconds at a time. As a curiosity if the nodes reach about 650 (at most) the browser tab crashes.

These results are very encouraging since with the filtering and depth limiting techniques employed, the upper limit we found sounds like an excessive amount of information to be rendering at one time (if using the application correctly) anyway.

5.2 STARTUP TIME

As mentioned before in order to improve rendering performance and keep the simplicity of the input format *Ulisses* need to perform some operations on the data when first loading it.

This metric is not critical to the functioning of the app itself but very heavy load times are obviously undesirable if the application is destined to be shown to an end user.

The test was performed by generating bigger and bigger datasets and feeding them to the graph while mesuring the time from calling the *drawGraph()* until the graph is actually rendered.

The previously described test yielded the following results:

Number of nodes/links	Load time
20000	Less than 1 second
40000	5 seconds
70000	10 seconds
100000	30 seconds
200000	90 seconds

Taking a look at these results we can see that the startup overhead is quite heavy. Still 100000 nodes/links confortably fits the description that the app should handle moderately sized graphs. What loading time is acceptable is a question that needs to be answered on a case by case basis.

With this in mind, this overhead does the job it is supposed to do and even with 200000 nodes loaded to the graph it runs very smoothly as long as the number of nodes currently visible on the display is manageable.

6 CONCLUSION

In conclusion to this thesis I would like to take a look at the state of the finished project with it's successes and challenges, evaluating both the graph visualizer and the turtle translator for their own merits as well as speaking a bit about the continued work that could be done to improve the current state of the app.

Considering the initially proposed approach, I'd say the the project was a resounding success, the final product includes a graph visualizer application with a well defined input format to access it's features, these same features allow for very diverse manipulation of the rendered data. This freedom to cater the input data to render very different types of data was one of the main specifications outlined at the start.

The other main focus for the visualizer was ease of navigation, through various techniques, the project manages to hit this mark as well, making sure that the information is intuitively available to even none developers using the app. Although making sure that these techniques take effect is the responsibility of the developer who generates the input format as a misuse of it can lead to hard to navigate graph.

On the subject of generating input graphs, the ontology translator does just that, it was developed as an add-on to the *UlissesNextGen* graph visualiser but is available as a completely separate package.

It takes on the difficult task of converting ontologies to a relevant and navigable graph. Due to the nature of this purpose the translator takes a very generalist approach to generating this graph, for simpler ontologies this is great and for the most part they can be handled very well, but as ontologies grow in specificity and/or complexity the disadvantages of this approach becomes apparent, as massive graphs with redundant information start to appear.

While the translator does what it is proposed to do there is an argument to be made against it's actual usefullness, for real use cases most of the users of the application will very much prefer to develop a translator tailored to their own ontology. Even so the provided turtle translator can serve as a strong foundation that can be refined to fit different situations.

With this overview of the project, we can conclude that the successes outweigh the challenges, as even where *Ulisses* struggles it gives space and tools to the users to provide their own solutions and ideas.

Taking a look at how *Ulisses* could be expanded in the future, there is a clear direction to take expanding variety of the rendering capabilities of the graph even further. This would obviously empower the application but should not be done at the expense of the relative simplicity of the input format.

In relation to the translator any expansion to it's current iteration would probably distance it from it's generalist intent. However in order to expand this facet of the project there could be added a solution to ease the development of other translators from ontologies to the input format, as a sort of framework for developers to cater to their own use cases.

BIBLIOGRAPHY

Tim Berners-Lee et al. Semantic web road map, 1998.

Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *ECOOP*, 2014.

Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D^3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011. doi: 10.1109/TVCG.2011.185.

Marek Dudáš, Steffen Lohmann, Vojtěch Svátek, and Dmitry Pavlov. Ontology visualization methods and tools: a survey of the state of the art. *The Knowledge Engineering Review*, 33, 07 2018. doi: 10.1017/S0269888918000073.

John H Gennari, Mark A Musen, Ray W Fergerson, William E Grosso, Monica Crubézy, Henrik Eriksson, Natalya F Noy, and Samson W Tu. The evolution of protégé: an environment for knowledge-based systems development. *International Journal of Human-Computer Studies*, 58(1):89–123, 2003. ISSN 1071-5819. doi: https://doi.org/10.1016/S1071-5819(02)00127-1. URL https://www.sciencedirect.com/science/article/pii/S1071581902001271.

Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing? *Int. J. Hum. Comput. Stud.*, 43:907–928, 1995.

Ian Horrocks, Peter Patel-Schneider, and Frank Harmelen. From shiq and rdf to owl: the making of a web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1:7–26, 07 2003. doi: 10.1016/j.websem.2003.07.001.

Giovani Librelotto, José Carlos Ramalho, and Pedro Rangel Henriques. Ulisses: Um navegador conceptual para topic maps. 01 2004.

Steffen Lohmann, Stefan Negru, Florian Haag, and Thomas Ertl. Visualizing ontologies with vowl. *Semantic Web*, 7:399–419, 05 2016. doi: 10.3233/SW-150200.

Mark Musen. The protégé project. *AI Matters*, 1:4–12, 06 2015. doi: 10.1145/2757001.2757003.

Hans-Jörg Schulz and H. Schumann. Visualizing graphs - a generalized view. volume 9, pages 166– 173, 08 2006. ISBN 0-7695-2602-0. doi: 10.1109/IV.2006.130.