

Title:

GraphHelper - Visualising & Exploring the Semantic Web using Intelligent Searching Tools

Theme:

Applied Intelligent Information Systems

Project Period:

02.09.2013 - 01.08.2014

Project Group:

IIS4-H106

Participant(s):

Heidi Munksgaard
Dines Madsen

Supervisor(s):

Dr. David L. Hicks
Jacobo Rouces
Michael Boelstoft Holte

Copies: 5

Page Numbers: 132

Date of Completion:

July 31, 2014

Abstract:

This report details the development of tools to assist users in efficiently searching through large **Resource Description Framework (RDF)** datasets. To accomplish this, the research group has developed a graph visualisation application called GraphHelper, which can represent **RDF**-graphs from both local files and remote repositories. In addition the research group proposes two algorithm for intelligent searching in **RDF**-data.

Interesting Graph is an **A***-like Breadth-First-Search algorithm, which to the best of the research groups knowledge, is a novel approach to exploring **RDF**-graphs. A user specifies search terms, based on which Interesting Graph searches the local neighbourhood of a designated node and returns the most promising subgraph to be visualised by GraphHelper. **Interesting Path** searches for relevant paths between two nodes, using a set of search terms specified by the user. This algorithm is an extension of previous work in the field. For both Interesting Graph and Paths, a set of experiments were performed in order to determine the **precision** of the algorithms using real-life data.

With this report comes a CD-ROM which contains documentation, source code, libraries and experimental results. GraphHelper and all libraries are open source.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Contents

Preface	vii
1 Project Description	3
1.1 Introduction	3
1.2 The Semantic Web	5
1.2.1 Resorce Description Framework	5
1.2.2 SPARQL	8
1.3 Problem Statement	16
1.4 Existing Solutions	17
1.4.1 Gruff	17
1.4.2 VisualRDF	17
1.4.3 RDF GRaph VIualization Tool	17
1.4.4 Welkin	18
1.4.5 GraphHelper	18
1.5 Language Selection	18
1.5.1 Java	18
1.5.2 Python	19
1.5.3 C#	19
1.5.4 Decision	20
1.6 Existing Libraries & Frameworks	20
1.6.1 RDF Data Management	20
1.6.2 Graph Visualisation	24
1.7 Frameworks Selected	25
2 Algorithms, Design & Implementation	27
2.1 Introduction	27
2.2 Software Overview	27
2.3 Jena Implementation	29
2.3.1 Data Management	30
2.3.2 Local File Models	32

2.3.3	Endpoints	35
2.4	JGraphX Implementation	39
2.4.1	Automatic Layout	41
2.4.2	Customization & Styles	43
2.4.3	Visualisation Implementation	44
2.4.4	VisGraph	44
2.4.5	Interesting Graph	50
2.4.6	Interesting Paths	51
2.5	Algorithms	54
2.5.1	Extracting Sub-Graphs	54
2.5.2	Interesting Graph	62
2.5.3	Interesting Paths	67
3	Experiments & Trials	75
3.1	Introduction	75
3.2	Measuring Interestingness	75
3.3	Interesting Paths	76
3.3.1	Question One	77
3.3.2	Question Two	77
3.3.3	Question Three	77
3.3.4	Question Four	77
3.3.5	Question Five	79
3.3.6	Reflections	81
3.4	Interesting Graph	81
3.4.1	Reflections	82
3.5	Parallel Bucket-Based Breadth First Search	84
3.5.1	Reflections	86
4	Discussion	87
4.1	Software Architecture Status	87
4.1.1	Graph Visualisation	87
4.1.2	Addressing Blank Nodes in Endpoints	88
4.2	Interesting Path & Graph	88
4.2.1	Semantic Improvements	88
4.3	Parallel Bucket-Based Breadth First Search	91
4.3.1	DataUnit Packing	91
4.3.2	Memory Management	91
4.3.3	Write-to-Disk	92
5	Conclusion	93
A	StringTemplate	97

B TDB	101
B.1 Concurrency	102
B.2 TDB Management	103
C Stanford Core NLP	105
D User Assisted Search	109
E Class Diagrams	111
E.1 Parallel Bucket-Based Breadth First Search	112
E.2 Visualisation Classes	113
E.3 Data Containers	114
E.4 RDFTree & WordScorer	115
E.5 Data Sources	116
E.6 Data Storage	117
Bibliography	119
Glossary	125
Acronyms	131

Preface

This Masters Thesis has been written by group IIS4-F-14 at Aalborg University Esbjerg in connection with the 9th and 10th semester Masters Project period from fall 2013 to summer 2014.

This project is based on the theme *Applied Intelligent Information Systems*, in which the emphasis is placed upon information retrieval and the [Semantic Web](#)

This report consists of five chapters and an appendix

Chapter One:

Sections [1.1](#) and [1.2](#) describes the project background and provides a brief primer on the [Semantic Web](#) and [SPARQL Protocol and RDF Query Language \(SPARQL\)](#). Section [1.3](#) formulates a problem statement as well as several project requirements. Sections [1.4](#) to [1.6](#) explore already existing solutions, both full applications, as well as frameworks and APIs.

Section [1.7](#) selects appropriate tools for the project to work with.

Chapter Two:

Section [2.2](#) presents the software architecture developed Sections [2.3](#) and [2.4](#) document how [Jena](#) and [JGraphX](#) is used throughout the project. Section [2.5](#) documents the intelligent search algorithms and remote sub-graph extraction algorithm developed by the project group.

Chapter Three:

Sections [3.2](#) to [3.4](#) discuss how to test the search algorithms developed, the results achieved, as well as reflections upon the results. Section [3.5](#) documents a run-time experiment on the sub-graph extracting algorithm devised.

Chapter Four:

Chapter [4](#) draws upon previous chapters, detailing issues encountered during devel-

opment, as well as expanding on future areas of research and improvements.

Chapter Five:

Chapter 5 summarizes across the project period and the results achieved, as well as reflecting on the future of research with regard to the Semantic Web.

The Appendix:

Chapters A to C are short introductions to libraries used throughout the project.

Chapter D describe a disambiguity feature used to assist users in searching.

Chapter E contains select class diagrams for the program developed.

References used in this report are written as [8] for referring to the full source in the bibliography or [45, p. 142-150] when referring to a specific area of a source. When referring to a specific section, the reference will look like this: section 3.2, while a table reference will look like this: table 3.1, a picture reference will look like this: fig. 1.3, a reference to a listing will look like this: listing 1.1, a reference to an equation will look like this: eq. (2.2) and references to an appendix looks like this: section E.3.

A CD-ROM containing the report, source code, libraries and experimental results is attached with the printed report.

Tools used for this project include: Oracle NetBeans 7.4, Microsoft Visio 2007, Sparx's Enterprise Architect 9

The code developed is dependent on the following frameworks and libraries: Java 7 64-bits, Apache Common IO 2.4, Apache Commons Collections 4.0, Google Guava 17.0, Jena 2.11, JGraphX 2.3.0.4, Stanford Core NLP 2014-01-04, String Template 4.0.8.

The research group would like to thank Dr. David L. Hicks, Ph.D student Jacobo Rouces and assistant Professor Michael Boelstoft Holte for their advice and guidance during the project.

In addition we would like to thank the following people for their support: Ph.D student Petar Durdevic, Study Secretary Britta Marie Jensen, Claus Hansen as well as our families and friends.

Aalborg University Esbjerg, July 31, 2014

Heidi Munksgaard
<hmunks05@student.aau.dk>

Dines Madsen
<dmadse09@student.aau.dk>

Project Description

1.1 Introduction

Since the birth of the Internet and the [World Wide Web \(WWW\)](#), the Web has been viewed as a web of documents, due to the vast amount of web-pages and documents it contains. This perception has changed due to the steady growth of the [Semantic Web](#) and advancement in information retrieval technology, with the Web now being considered more as a web of data-points. Data is now no longer constrained to be situated in a monolithic block on web-pages but can be collected and merged into ontological units in which data is being categorized by type and relationships rather than location in a given document.

Data contained in the [Semantic Web](#) is stored in directed labelled graphs to show not only the data itself, but also properties of the data and its relations to other relevant data.

This new type of data abstraction provides [Semantic Web](#) users with an immense amount of information, which can not easily be navigated in a feasible manner using traditional tools such as [Structured Query Language \(SQL\)](#) or other traditional information retrieval tools. What is needed instead are tools that simplifies the structure of the information displayed to the user. Several tools have been developed for solving the issue of abstracting non relevant data away from the user. These solutions are based on three different methodologies:

- SPARQL
- Hierarchical structure
- Graphical visualisation

The [SPARQL](#) language is to the [Semantic Web](#), what [SQL](#) is to relational databases. With [SPARQL](#) queries it is possible for the user to retrieve a sub-set of the whole graph, and have it displayed in formatted text. In order for the user

to interpret the sub-sets generated with **SPARQL**, some prior knowledge is usually required. For a user, without a relevant technical background, the learning curve for using **SPARQL** efficiently can be steep, which makes graph navigation with **SPARQL** unrealistic for a layperson.

Another way of abstracting away information from the user is to display information in a hierarchical structure, in which the information is divided into categories which can be expanded out to sub-categories. This way, the user can choose a main category, and then via mouse clicking, expand this main category into sub-categories in a tree-like manner, in much the same way as navigating the folder-structure on a computer. One application which has implemented hierarchical structure model is **gFacet** [34]. Besides the hierarchical structure, which has been implemented to provide the user with a graph-like user interface, this application provides the user with different options for viewing the data, called facets. This way different interrelations can be displayed to the user dependent on the facet chosen.

Since **gFacet** provides the user with a **Graphical User Interface (GUI)** the learning curve is not nearly as steep to users without prior knowledge compared with **SPARQL**. But **gFacet** does not provide the user with a sub-graph showing actual connections between entities in the graph, but rather aggregated connections between entities.

In the third method, graphical visualisation, data is laid-out as nodes with connections and the prior knowledge required for using a graph visualisation systems is comparable to using systems like **gFacet**. Issues arise with the use of tools displaying all connections between all entities in a graph. There can be thousands or even hundreds of thousands of connections between a given entity and all other entities in the graph. Such a graph is full of clutter and will quickly confuse a user more than it will aid them, when trying to discover relevant connections between entities in a graph.

This project was inspired by researchers associated with the **early Pursuit against Organized crime using envirOnmental scanning, the Law and IntelligenCE** systems (**ePOOLICE**) project. The **ePOOLICE** project is a joint European research project aimed at merging large numbers of public information sources into a predictive data model aimed towards early detection of organized crime. This way subject-matter-experts are provided with advanced tools for making predictions about organized crime.

As the subject-matter-experts do not necessarily posses the necessary technical knowledge and skills to efficiently navigate large data sets, it was suggested that the research group should explore options of providing users with a graphical visualisation tool, that has the means to simplify searching and interpreting graphs. This way the discovery of relevant information will be more efficient for the users.

1.2 The Semantic Web

The early 1990's saw a boom in information technology with the invention of the [WWW](#), which created a vast network of web-pages, connected by [Uniform Resource Identifier \(URI\)](#) technology. The [WWW](#) quickly began accumulating large amounts of information, that was easily parse-able by computers, and readable by humans. However it proved difficult for machines to automatically extract information from web-pages, due to the implicit nature with which the information is presented. Many different methods have been developed for [Data mining](#) information from the [WWW](#), but most rely on heuristics or probabilistic approaches, which can produce unreliable results. In 2001 Dr. Tim Berners-Lee[3] published an article, describing a scenario in which computers effortlessly could explore the [WWW](#), extracting information and making inferences about said information with minimal human intervention. This new network, which Dr. Berners-Lee called the [Semantic Web](#), would not be a replacement of the [WWW](#), but rather provide an additional layer of information. The goal for the [Semantic Web](#) was to be machine-readable, and more importantly, sufficiently semantically rich for machines to draw inferences, but also flexible enough so that a wide variety of information could be described. The system proposed by Dr. Berners-Lee was named the [RDF](#)

1.2.1 Resorce Description Framework

[RDF](#) is a recommendation put forth by the [World Wide Web Consortium \(W3C\)](#) in 1999[9], with a revised version published in 2004[8]. The [RDF](#) is a [W3C](#) recommendation, used for decomposing knowledge into little pieces. These little pieces are called resources.

A resource is a description of a given item, which could be a person, a city or another item described with a noun. The resource is identified with a unique [URI](#), a literal or a blank node. A literal is a string denoting a certain value, for example the number "ten" or the city "Berlin". A blank node is a way of describing an entity that cannot directly be identified. An example could be: "George buys a present for his friend". In this case "friend" is a node without a unique description, since George can have several friends. This type of resource is also called an anonymous resource. [RDF](#) does not have a predefined storage format, but instead a host of different serialization-forms that can be used, such as [Resource Description Framework Extensible Markup Language \(RDF/XML\)](#), [Turtle](#), [N-Triples](#), [Notation 3](#), [OWL Web Ontology Language](#) and [OWL 2 Web Ontology Language](#). [RDF](#) is specifically kept abstract so that it has the power to describe a wide range of topics. The relation between the resources is described as a directed labelled graph, which is also called an [RDF](#) graph. Each resource is a node in this graph and the edge between them is a predicate. Since an [RDF](#) graph is directed the node from which the edge goes out, is called a subject, while the node to which the edge goes in is called an object. So the [RDF](#) framework describes a model by which a [directed graph](#) can be represented as a set of triples, with each triple containing a *subject*, *predicate* and *object*, commonly

written as:

$$<\text{subject}, \text{predicate}, \text{object}>$$

A triple can be used to express a wide range of relations such as: “ John Doe owns a VW golf”, as illustrated by fig. 1.1

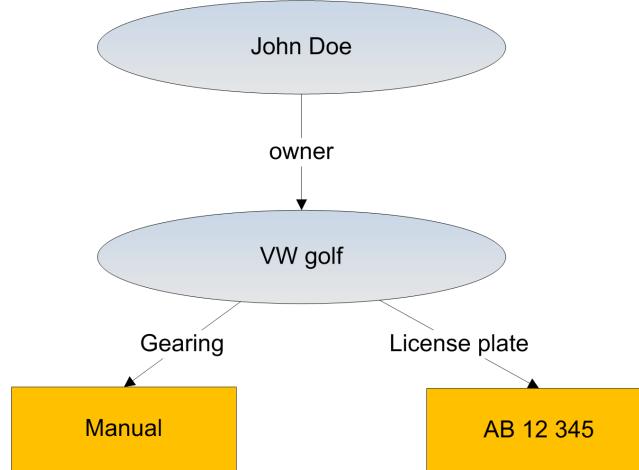


Figure 1.1: A simple directed graph, illustrating *John Doe*'s ownership of a *VW golf*

Figure 1.1 illustrates the ownership of a VW golf, by a John Doe. The car has two properties, namely a license plate and manual gearing. Translating the graph into triples, would result in the following:

$$\begin{aligned} &<\text{John Doe}, \text{ownsCar}, \text{VW golf}> \\ &<\text{VW golf}, \text{License plate}, \text{AB 12 345}> \\ &<\text{VW golf}, \text{Gearing}, \text{Manual}> \end{aligned}$$

While this representation does have considerable flexibility, it does suffer from ambiguity. What nationality is the *license plate*? It might be possible to deduce the nationality based on the format, but this would require domain specific knowledge, and would therefore introduce uncertainty into the graph. In order to avoid this ambiguity, **RDF** uses **URIs** to uniquely identify resource types. By introducing **URI**'s, the issues faced in fig. 1.1 have been solved in fig. 1.2

The different definitions for *Gearing* will not cause an issue, as the **URI**'s uniquely define where to find their individual definitions. It is also worth noting that John Doe does not have to create all necessary definitions, but instead can rely on other definitions, in this case from www.VM.com and www.centurion.dk, to define aspects.

Not every field is an **URI**. The three boxes *Manual*, *AB 12 345* and *7* are called *literals*. Two types of literals are possible

Plain literal are plain strings with an optional language tag

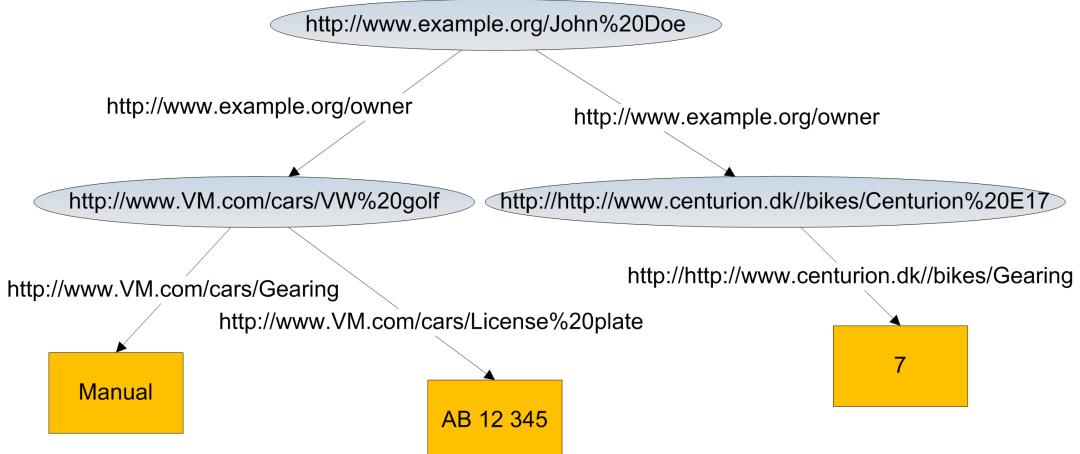


Figure 1.2: John Doe has expanded his possessions with a bike

Typed literal are strings bundled with datatype information. **RDF** does not provide explicit datatype information, but rather defers the definitions of types to developers

Literals can only be used as *objects*, as it does not make conceptual sense to have a subject or predicate literal.

1.2.2 SPARQL

SPARQL[31] is a query language used for querying **RDF** graphs either situated on the web, in a file or in an RDF-triplestore. **SPARQL** is a descriptive language which allows for complex descriptions of graph-patterns. For users of **SQL**, many of the keywords and base structures of **SPARQL** will be familiar. The newest version of **SPARQL** is v1.1 which was published 21-03-2013 and the recommendations are developed and managed by **W3C**.

There are four types of basic patterns which **SPARQL** supports

- Triple patterns
- Conjunctions
- Disjunctions
- Optional patterns

The four types of patterns are mixed in different ways in order to achieve the desired pattern.

Triple Pattern

The most common pattern which describes the basic **RDF** pattern of *Subject-Predicate-Object*. Listing 1.1 illustrates a simple triple pattern search

Listing 1.1: An example of a SPARQL triple pattern query

```
PREFIX rdf:<http://www.w3.org/2001/vcard-rdf/3.0#>

SELECT * WHERE
{
  ?x rdf:FN ?fname
}
```

The output resulting listing 1.1 query will be the subject x and the object $fname$ of type literal, with the predicate $rdf:FN$. This predicate is an **RDF** standard entity, and represents the full name of an entity. Figure 1.3 shows a simple triple pattern that would be returned by listing 1.1.

Conjunction Pattern

Conjunction patterns are two or more triple patterns which when taken together describe a more complex pattern. An example of a simple conjunction query can be seen in listing 1.2



Figure 1.3: An example of a **SPARQL** triple pattern result

Listing 1.2: An example of a **SPARQL Conjunction Query**

```
PREFIX foaf:<http://xmlns.com/foaf/0.1>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE
{
    ?person rdf:type foaf:Person .
    ?person foaf:name ?name .
    ?person foaf:mbox ?email .
}
```

The resulting output from this query, illustrated by fig. 1.4 will be the objects of a subject of the `rdf:Type Person`. The objects: `name` and `email` are literals containing the name and email address of the subject. This query is conjunctive, which is equivalent to a logic *AND* since it will only output a result for vertices where all three properties are present.

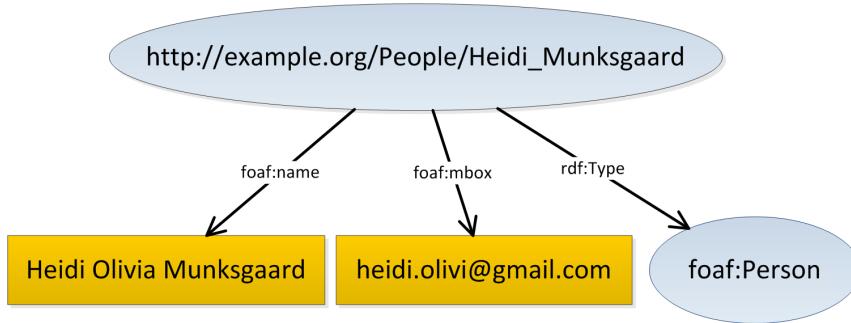


Figure 1.4: An example of a **SPARQL** intersection pattern

Disjunction Pattern

Disjunction patterns merges result-sets of sub-queries. An example of a disjunction query is shown in listing 1.3

Listing 1.3: An example of a **SPARQL** Disjunction Query

```
PREFIX dc10: <http://purl.org/dc/elements/1.0/>
PREFIX dc11: <http://purl.org/dc/elements/1.1/>

SELECT * WHERE
{
  { ?book dc10:title ?title }
  UNION
  { ?book dc11:title ?title }
}
```

The resulting output from this query, which is illustrated in fig. 1.5, will be the object literals equal to the title given by the predicate dc10 or dc11. The query

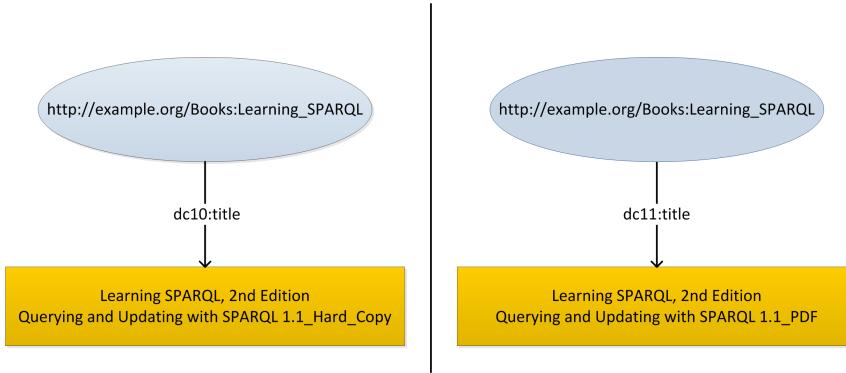


Figure 1.5: An example of a **SPARQL** disjunction pattern

uses the *UNION* keyword, which is equivalent to a logic *OR* and there will therefore be looking for books with either dc10:title or dc11:title and merging the sets. If an subject has both a dc10:title and dc11:title, the subject will be outputted twice.

Optional Pattern

Optional patterns allow for flexibility in **SPARQL** queries. An example of an optional query can be seen in listing 1.4.

Listing 1.4: An example of a **SPARQL** optional query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT * WHERE
```

```
{
    ?person foaf:name ?name .
    OPTIONAL (?person foaf:depiction ?depiction) .
}
```

As can be seen on fig. 1.6 the resulting output from this query will be the name of a person who is the friend of a friend of a given entity. If there is a depiction of the person from the result set, this depiction will also be output. If no such depiction exists for an entity in the result set, only the name will be displayed.

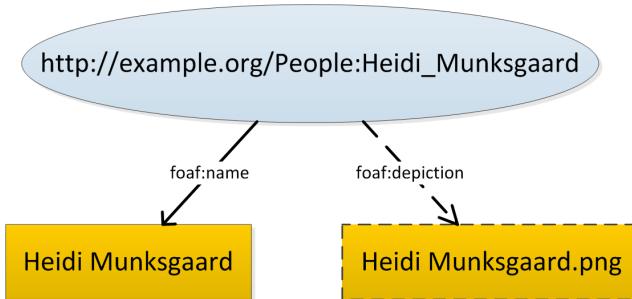


Figure 1.6: An example of a [SPARQL](#) optional pattern

SPARQL keywords

The four [SPARQL](#) keywords used for the actual data extraction are:

SELECT - is equivalent to the [SQL SELECT](#), meaning that out of the full set retrieved elements, only the specified values are returned.

CONSTRUCT - Creates a sub-graph consisting of the triples from those found in the WHERE clause. CONSTRUCT is often used to make implicit information explicit.

ASK - Asks whether a graph contains a specific pattern. Returns true or false only

DESCRIBE - Describes a specific node. There is no specification for *how* a node should be described in the [SPARQL](#) 1.1 standard. It can therefore be somewhat unreliable when working with remote [Triplestore](#), unless the [Triplestore](#) documentation has been read

In addition to the query patterns listed above, [SPARQL](#) also provides keywords for arranging the output in various ways:

ORDER BY ?value - Orders the result by ?value in ascending(default) or descending order

GROUP BY ?value - Groups results by ?value

LIMIT k - Returns the first k entries or less of the result set

OFFSET k - skips the first k results. Often used in combination with LIMIT

PREFIX - Defines prefixes for [URIs](#) for later use in the query. This is done in order to make queries shorter and easier to read for humans.

The last group of [SPARQL](#) keywords are used for expanding or limiting the amount of data fetched when using the keywords in the uppermost listing.

FILTER - Removes entities which fulfill the requirements in the FILTER expression

UNION - Is used in disjunction queries and merges two result sets.

OPTIONAL - Is used if properties belonging to a group of entities that are non-uniform.

The following examples highlight usage of the terms presented so far. As an example, a [Turtle](#) file which can be seen in listing 1.5, represents people and their family relations, those being their names, gender and parents. An [RDF](#) sub-graph can be constructed as well as data, which was initially implicit, can be made explicit using *CONSTRUCT*. An example of the use of the *CONSTRUCT* keyword can be seen on listing 1.6, which will construct an [RDF](#) graph showing the grandfather and grandchildren in the family described in the [Turtle](#) file.

Listing 1.5: [Turtle](#) file describing human relations

```
@prefix : <http://www.snee.com/ns/demo#> .  
  
:jane :hasParent :gene .  
:gene :hasParent :pat ;  
      :gender   :female .  
:joan :hasParent :pat ;  
      :gender   :female .  
:pat  :gender   :male .  
:mike :hasParent :joan .
```

Listing 1.6: SPARQL Construct query

```
PREFIX : <http://www.snee.com/ns/demo#>

CONSTRUCT { ?p :hasGrandfather ?g . } WHERE
{
    ?p      :hasParent ?parent .
    ?parent :hasParent ?g .
    ?g      :gender      :male .
}
```

An example of the output from the *CONSTRUCT* query on listing 1.6 can be seen on fig. 1.7

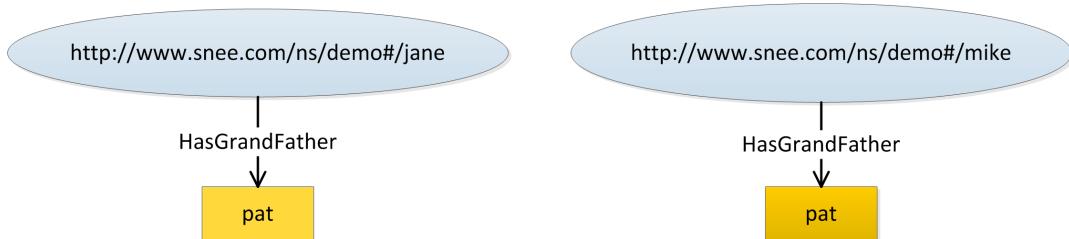


Figure 1.7: An example of inferring implicit data to make it explicit

As mentioned above another use for the *CONSTRUCT* feature is to create an **RDF** sub-graph which can be constructed from a given result set. An example of this usage can be seen on listing 1.7. With this *CONSTRUCT* query it is possible to construct a sub-graph with subjects and objects with the corresponding literals “Einstein”, “Bohr” and “Faraday” only, but with any given predicates between these nodes.

Listing 1.7: SPARQL CONSTRUCT query for creating a sub-graph

```
PREFIX : <http://stackoverflow.com/q/20840883/1281433/>

CONSTRUCT {
  ?s ?p ?o
}
WHERE
{
  VALUES ?s { :"Einstein" :"Bohr" :"Faraday" }
  VALUES ?o { :"Einstein" :"Bohr" :"Faraday" }
  ?s ?p ?o
}
```

The *ASK* query is used when a boolean result is required. The example of such a query taken from the book: *Learning SPARQL by Bob DuCharme*[11, p.119], can be seen on listing 1.8. The output from the query shown on listing 1.8 will be either *true* or *false* depending on whether a subject *s* has an object *city* with an invalid *URI*

Listing 1.8: SPARQL ASK query return boolean results

```
PREFIX dm: <http://learningsparql.com/ns/demo#>
ASK WHERE
{
  ?s dm:location ?city .
  FILTER(!isURI(?city)) .
}
```

The *FILTER* keyword is used when wishing to filter out specific items from a result set. An example of the use of the *FILTER* keyword can be seen in listing 1.9. This query will select all entities of cities in Texas with a population larger than 50.000. In addition it will select the metro population, which is a sum of the population of adjacent cities, if it exists. Besides the filtering the query also contains the keyword *ORDER BY*, which will put the resulting data set in a given order. In this case the cities are ordered in descending order according to population size.

Listing 1.9: SPARQL Optional Query with filter and order by

```

1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3 PREFIX dbp: <http://dbpedia.org/ontology/>
4
5 SELECT * WHERE {
6   ?city rdf:type
7     <http://dbpedia.org/class/yago/CitiesInTexas>;
8   ?city dbp:populationTotal ?popTotal .
9   OPTIONAL (?city dbp:populationMetro ?popMetro) .
10  FILTER (?popTotal > 50000)
11 }
11 ORDER BY desc ?popTotal

```

The *GROUP BY* keyword is used in **SPARQL** in the same way as it is used in **SQL**, namely for grouping certain items together. This example of a query using the *GROUP BY* taken from the book: *Learning SPARQL by Bob DuCharme*[11, p. 94], can be seen on table 1.1. This query will select all items matching a given description. The amount value contained in the triples with the given description name will be summed together and stored in a variable called *amount* with the use of the *SUM* function summing all the values from *e:amount* and storing them in the variable *mealTotal* with the use of the *AS* function. The *SUM* and *AS* functions are redundant to the corresponding functions in **SQL**. When the given data has been processed it will be grouped by description showing the *mealTotal* value corresponding to each group. As an example of an output running the query in listing 1.10 can be seen on table 1.1.

Listing 1.10: SPARQL query using the GROUP BY keyword

```

PREFIX e: <http://learningsparql.com/ns/expenses#>
SELECT ?description (SUM(?amount) AS ?mealTotal)
WHERE
{
  ?meal e:description ?description ;
  e:amount ?amount .
}
GROUP BY ?description

```

description	mealTotal
dinner	84.80
lunch	30.58
breakfast	17.50

Table 1.1: Output from the [SPARQL](#) query from listing 1.10.

The *LIMIT* keyword is used when wanting to limit the result set to a given amount of triples, meaning that if you set *LIMIT* to be equal to 20, only the 20 first results will be displayed for you.

1.3 Problem Statement

As it is not feasible for laypeople to effectively navigate the [Semantic Web](#) with the current set of visualisation tools, the project group puts forth the following problem statement:

How to develop a graph visualisation application with tools which assist end-users with the discovery of relevant information from Semantic Web sources

As this application will be used for research purposes, the developed code needs to be open-sourced.

The resulting application can be broken down into a set of sub-requirements

File Format and Triplestore Support

The application should be able to load and save models from widely used graph file formats, such as:

- RDF/XML
- N3
- Turtle
- N-Triples
- OWL
- OWL 2

In addition to accepting data from common file formats, the application should also be able to interface with [Triplestores](#) for storage and retrieval of triples. This interface should be accessible both programmatically, as well as through human-entered

SPARQL queries.

Graph Visualisation

The application should be able to render large complex graphs, containing nodes, edges and literals. Support for colour coding of edges and nodes/literals is necessary in order to produce more human-readable graphs. It should be possible to spawn context menus from objects on the graph, in order to manipulate or explore the graph. Graphs should be exportable into common image formats.

Intelligent Exploration Tools

The application should assist a user in exploring and searching through complex RDF-Graphs. The application should allow a user to filter out unwanted relations as well as provide a method for specifying topics of interest. The results of these explorations should be visualisable.

1.4 Existing Solutions

In order to speed up development, it will be desirable to make use of already existing solutions, although some modifications most likely will be needed.

1.4.1 Gruff

Gruff[36] is a closed-source graph visualiser application, developed by Franz Inc. Gruff is used for visualisation of sub-graphs extracted from AllegroGraph, a Triple-store also developed by Franz Inc. As Gruff does not fulfill the requirements of being open-sourced and applicable to multiple types of triple stores, it is not considered a valid candidate for further exploration.

1.4.2 VisualRDF

This application was developed by Alvaro Graves[19] and published on GitHub as a visualisation tool for RDF graphs. VisualRDF shows simple RDF-graphs in a website, and allows a user to drag and move nodes in the graph. As there has been no updates on this project for more than six months, the documentation is poor, and the implementation very rudimentary, this application is not consideration as a candidate for the project solution.

1.4.3 RDF GRAph VIualization Tool

The RDF GRAph VIualization Tool (RDF-Gravity)[17] tool is modelled on top of Java Universal Network/Graph Framework (JUNG) a Java library, which provides an Application Programming Interface (API) for visualisation data. RDF-Gravity visualiser both RDF graphs and OWL Web Ontology Language graphs. RDF-Gravity has a GUI which gives the user the option of merging multiple files into one graph,

as well as offering zoom, rendering and selection of areas in a graph. The software also provides the user with a selection of filtering tools for simplifying views of a given graph, as well as providing the user with full text search, with the use of an [RDF Data Query Language \(RDQL\)](#). This tool will fulfill most of the requirements stated in the requirement specifications, but the license is not open-source and [RDF-Gravity](#) is therefore not a candidate.

1.4.4 Welkin

[Welkin](#)[46] is a graph visualisation tool used to find the density of very large graphs, and does not show the labels of properties. Nor does it enable users to extract single nodes or paths, which is a requirement for this project solution. [Welkin](#) is therefore not a candidate for further work.

1.4.5 GraphHelper

As no suitable application to expand upon was found, the research group has decided to create an application in order to address the requirements set forth in section 1.3. This application will be named *GraphHelper*, and in the following sections [APIs](#) and frameworks which might speed up development will be discussed.

1.5 Language Selection

In order to make an informed choice with regard to programming language for the project, the following requirements are set forth:

1. Open Source - As the project product will be used for research purposes, an open source requirement will ensure that modifications and expansions can be implemented, without legal issues arising.
2. Multi-platform - The product should be executable across all major operating systems: Windows, Mac OS and Linux.
3. Good library ecology - Implementing every feature from the bottom up, would be exhaustive. Good open source libraries will speed up development, and is library ecology is therefore an important aspect

As the set of programming languages available is incredible large, we limit the scope to three of the major open source object oriented languages, namely Java, Python and C#

1.5.1 Java

[Java](#)[50] is one of the major object oriented programming languages today. It is open source, and is supported on all the major operating systems through the [Java](#)

Virtual Machine (JVM). [Java](#) has a strong open source community attached, which is reflected in the wide range of libraries available to developers.

For graph manipulation, there are several minor projects that handle either parsing of serialized files, or performing inferences, but the two major projects within the field of [RDF](#)-graphs, are the [Jena](#)-[37] and [Sesame](#)[49] frameworks. [Jena](#) and [Sesame](#) are open source frameworks, aimed towards providing tools for the task of working with [RDF](#)-graphs. Both frameworks contain tools for parsing and serializing graphs in multiple different formats, as well as in-memory [Triplestore](#) for manipulations of graphs. [SPARQL](#) queries can be performed on in-memory models with both [Jena](#) and [Sesame](#) as well as against web-based endpoints.

For visualisation of graphs, [Java](#) has a wide range of libraries, some more developed than others. As interaction with the graph is necessary, we will ignore libraries which generate static images of graphs, and instead focus on those libraries which provide access to elements in a plotted graph. Most of the major frameworks, such as JGraphx[21], JUNG[22] and Prefuse[24] allow for the creation of [Swing](#) frames which can be embedded into graphical interfaces. Each of the three libraries mentioned support a various kinds of graphs, including directed, as well as methods for automatic placement of nodes.

1.5.2 Python

[Python](#)[13] is an open source programming language, that supports a number of programming paradigms, including object oriented. [Python](#) has native support in most Linux distributions, and has various implementations on Microsoft Windows and Apple Mac OS. [Python](#) has several libraries aimed at working with [RDF](#)-graphs.

[RDFLib](#)[26] is one of the major [RDF](#)-graph libraries, containing a host of tools. [RDFLib](#) can parse common graph formats, such as [RDF/XML](#), N3, Turtle, TriX and more. [SPARQL](#) queries can be executed against in-memory models, as well as against certain external triplestores. [RDFLib](#) also has support for storing triples within common relational databases, such as MySQL, PostgreSQL and SQLite.

[RDFlib](#) has support for OWL 2's RL profile, extending normal [RDF](#)-graphs with OWL triples, as well as adding support for OWL serialisation format. Reasoning across OWL sets can be performed using FuXi[15]

There are several libraries for visualising various types of graphs. NetworkX[23] allows not only for visualisation, but also contains common network analysis tools, such as shortest path, betweenness and centrality algorithms. igraph[35] is another graph library, for Python, R and Ruby. As with NetworkX, igraph contains a set of tools for network analysis as well as visualising.

1.5.3 C#

C# is a relatively young programming languages, which supports imperative, objective and functional programming paradigms. C# has native support on most Windows platform, with support for Linux and Mac OS via The Mono Project[55].

For graph manipulation, [Jena](#) is available as a ported version for .NET[14]. The .NET version of [Jena](#) was last updated 19.12.2010, is therefore by no means state-of-the-art. [Sesame](#) has also been ported to .NET and has been named [dotSesame](#)[49]. Dotsesame is a significantly more recent porting of [Sesame](#), with the latest version being generated late Jan. 2014.

C# does not have as wide a selection of graph visualisation frameworks as [Java](#) or Python. However GraphX[12], Microsoft Automatic Graph Layout[47] and Graph#[51] all offer the basic visualisation, layout and manipulation tools necessary for this project.

1.5.4 Decision

Looking at the three requirements set forth for selecting a language, all three languages meeting requirement 1 and 2. While C# does have ported versions of [Sesame](#) and [Jena](#), there is some concern regarding rates of updates and support, especially when considering the pace at which [Semantic Web](#) applications and frameworks are evolving. RDFLib, [Jena](#) and [Sesame](#) all have sizeable groups of both users and developers, making both [Java](#) and Python viable options with regard to framework selection. As the project group already has experience with [Java](#), the implementation language will be [Java](#).

1.6 Existing Libraries & Frameworks

This section describes existing libraries, frameworks and APIs, which could at least partially solve the problem stated for this project. The section is divided into existing solutions for the logical data management of the problem stated for this project and the visualisation of semantic data.

1.6.1 RDF Data Management

In this subsection the research group has looked at existing solutions for solving the processing of [RDF](#) triples and graphs in order to find the most appropriate tools for fulfilling the requirement specifications for the logical data management of the project solution. The two existing solutions described in this subsection are. The [Jena](#) Framework and The [OpenRDF Sesame Framework](#). These two frameworks have been chosen because they are open-sourced, they are still supported and the software has been developed on so it has reached an appropriate maturity. This means that most bug issues have been solved and the documentation makes the implementation of these frameworks relatively simple.

Jena Framework

The [Jena](#) framework is written in [Java](#), with [Jenas](#) architecture diagram shown in fig. 1.8. The major three components, *RDF API*, *Ontology API* and *SPARQL API*

are the core APIs of Jena. In addition to these components, Jena has parsers in order to read and write various RDF file formats. Jena supports several forms of model management, from in-memory models, to simple disk-based Triplestores(TDB) and web-interfaceable Triplestores(Fuseki).

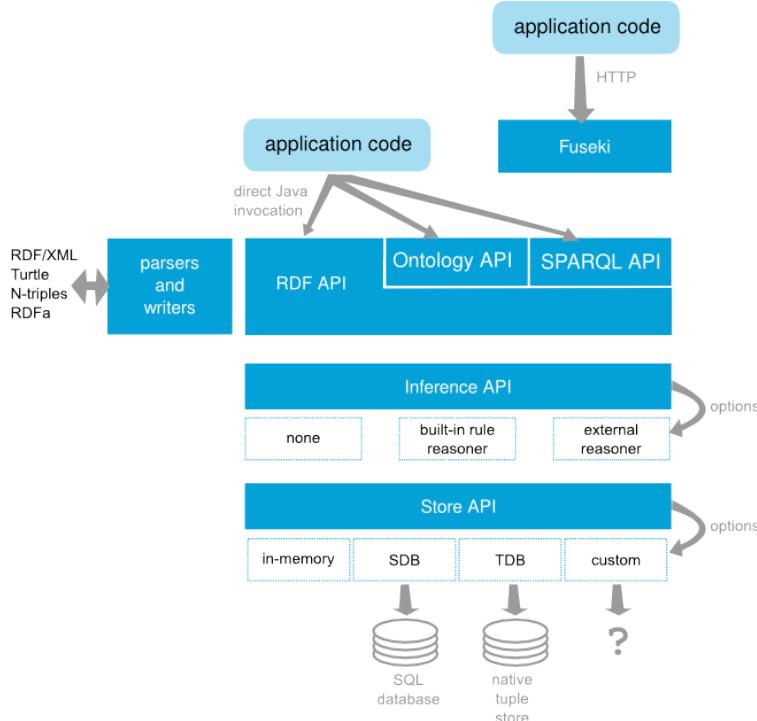


Figure 1.8: The Jena Architecture [41]

Jena RDF API

The Jena RDF API can be used to create and manipulate RDF graphs with the use of a class called *Model*. This class can be used to create the nodes and edges, which makes up a graph. The Jena RDF API can be reached through Java application code alongside the Jena OWL Web Ontology Language API and the Jena SPARQL API. This way SPARQL queries can be invoked and OWL Web Ontology Language inferences can be drawn on a given model in the same file.

Jena Ontology API

The Jena Ontology API is used for extracting previously unknown knowledge from the triples in a given RDF graph. Since this API will not be used in this project solution, this API will not be explained any further.

Jena SPARQL API

As shown in the architecture diagram in fig. 1.8 the [Jena SPARQL API](#) can be invoked from the command-line through pre-compiled executables, as well as through the [Java API](#). This means that a [SPARQL](#) query can be run directly on the model from the [Jena RDF API](#).

Inference API

The [Jena Inference API](#) is a generic inference subsystem, to which a range of inference engines or reasoners can be plugged in. These types of engines are used to extract additional information out of an [RDF](#) graph in addition to the information extracted with classes from the [APIs](#) from the upper most layers. Since this [API](#) is not used in the project solution, this [API](#) will not be explained any further.

Storage API

The [Jena Storage API](#) provides the interface for storing [RDF](#) triples. The file formats, which are supported for storage and fetching through the [Jena Storage API](#) include:

- [RDF/XML](#)
- [Notation 3](#)
- [Turtle](#)
- [OWL Web Ontology Language](#)
- [OWL 2 Web Ontology Language](#)

When fetching triples from an end-point triple store such as [DBpedia](#) it will often suffice to store these triples in a *Model* in-memory. But if the set of triples is too big to be stored in-memory, the [Jena Storage API](#) also provides other storage abstractions such as [SDB](#), [TDB](#) and if needed, custom made databases.

Fuseki

Besides the [APIs](#) just described, [Jena](#) also provides software for setting up a [RDF Triplestore](#) server. This software is called [Fuseki](#) and it can be accessed through the application code made for the three uppermost [APIs](#) in the [Jena](#) Framework as well as in a stand-alone customized [Java](#) application.

OpenRDF Sesame Framework

The [OpenRDF Sesame Framework](#) is like the [Jena](#) Framework written in [Java](#). As can be seen on fig. 1.9 the [APIs](#) are similar to the ones in the [Jena](#) architecture, only with the layers in reverse order. In this section the components making up The [OpenRDF Sesame Framework](#) will be described briefly.

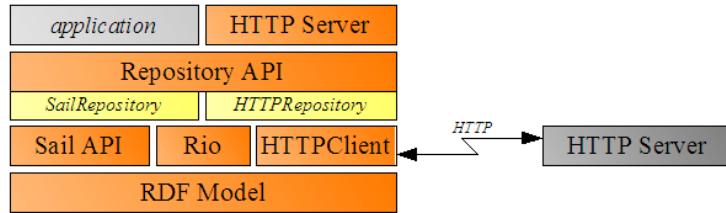


Figure 1.9: The OpenRDF Sesame Architecture

RDF Model

In the **RDF Model**, which is the lowest layer in the *OpenRDF Sesame Framework*, as can be seen on fig. 1.9, all basic **RDF** entities and methods are defined. The entities include URIs, blank nodes, literals and statements while the methods include processing of models and their entities.

Storage And Inference Layer API

As the name implies the **Storage And Inference Layer (Sail) API** is used for storing files or to extract additional information out of an **RDF** graph. The storage abstractions can be either in-memory or disk-based. The **Sail API** supports the same file formats as the **Jena Storage API**.

RDF/IO

RDF/IO (Rio) contains parsers for reading in and writing out **RDF** statements into and out of files.

HTTPClient API

This **API** is used for communication with an **HTTP** server which is not necessarily a part of the *Sesame* Framework. The **HTTPClient API** can therefore be used for generic **HTTP** communication with an external server **Triplestore**.

Repository API

This **API** It contains methods for uploading, manipulating, querying and processing data. The **Repository API** is implemented in a number of different classes such as the *SailRepository* and the *HTTPRepository*. The *SailRepository* translates calls into *Sail* objects while the *HTTPRepository* will process the calls into **HTTP** objects.

HTTP Server

The **HTTP Server** is in the top of the software layer of the *OpenRDF Sesame Framework*. In the same way that the **Fuseki** server can be accessed through application code in the **Jena** Framework, the **HTTP Server** for The *OpenRDF Sesame framework* can be accessed through application code and as well as through the **Repository API** and directly through the **HTTPClient**.

1.6.2 Graph Visualisation

There are several open-source graph visualisation framework available. In this section the most relevant frameworks will be discussed, although this is by no means a thorough analysis of all frameworks.

JGraphX Framework

The [JGraphX](#) Framework[21] is a [Java Swing](#) library originally developed under the name *JGraph*, aimed towards JavaScript. *JGraph* was forked and renamed to [JGraphX](#) under a [Berkeley Software Distribution](#) license. [JGraphX](#) offers a rich set of features such as:

- Automatic layout algorithms
- Event-based actions
- Context menu support
- Various styles for creation of custom look and feel of graphs

[JGraphX](#) is still actively being developed on, and would be an idea candidate for this project. The framework supports color coding for edges, nodes and literals and enables spawning of context menus from objects on the graph. It also supports exporting of graphs in a range of file formats(PNG, JPEG, BMP and more).

JUNG

[JUNG](#)[22] is not only a graph visualisation framework, but also contains tools for network analysis such as calculating flow-rates and shortest-paths. [JUNG](#) is written in [Java](#) and has support for

- Context menu support
- Automatic layout algorithms
- Event-based actions
- Custom rendering through overwriting of object forms

Graphs can be exported to images to most common image formats in various qualities. [JUNG](#) has seen limited development. At the time of writing, the last version was published 29-05-2013, which does produce some concern about the longevity of [JUNG](#).

Prefuse

Prefuse[24] is a visualisation framework for a wide range of applications, such as tables, graphs, trees and more. It is licensed under [Berkeley Software Distribution](#) and supports the following features

- Context menu support
- Automatic layout algorithms
- Event-based actions
- Various styles for creation of custom look and feel of graphs

Prefuse uses [Swing](#) components and contains a host of more advanced features such as animation and integrated search. Prefuse has seen limited development in recent time.

1.7 Frameworks Selected

After exploring the existing libraries, frameworks and [APIs](#) in this section, the choice has fallen upon the [Jena](#) framework for handling [RDF](#) data and [JGraphX](#) framework has been chosen for visualisation. They are both open-sourced and still actively being developed on. The [Jena](#) Framework supports all file formats listed in the sub requirements in section [1.3](#), as well interfacing with [Triplestores](#) both programmatically and through human-entered [SPARQL](#) queries.

[JGraphX](#) Framework fulfills the sub-requirements of being able to render large, complex graph, containing nodes edges and literals. The framework also supports color coding of edges, nodes and literals, as well as enabling spawning of context menus from objects on the graph. [JGraphX](#) also has support for exporting graphs as images. While the frameworks selected fulfill a good deal of the sub requirements, the problem of assisting a user in exploring and searching for specific graphs or paths based on topics of interest as well as these features remains unsolved, as far as existing libraries, frameworks and [APIs](#) are concerned. The solution to these problems will therefore be the focus of a large part of this project.

Chapter 2

Algorithms, Design & Implementation

2.1 Introduction

This chapter describes how *GraphHelper* is implemented. Section 2.2 presents the software architecture designed, as well as the rational behind it. Sections 2.3 and 2.4 describes how [Jena](#) and [JGraphX](#) is used throughout *GraphHelper*. Section 2.5 explores the algorithms implemented within *GraphHelper* for intelligent searching, as well as for efficient extraction of sub-graphs from remote endpoints.

2.2 Software Overview

Software is often developed and expanded upon at incredible rates, as new features are added and old bugs are corrected, especially within areas where active research is being undertaken. For this project, the [Jena](#) framework was selected as the backbone for [RDF](#) data storage and manipulation. Due to the rapid development of frameworks, it might later be necessary to switch to a different framework in order to fulfill new requirements. If [Jena](#) is directly integrated into the software developed, with hooks going to various [GUI](#) components, it will be a prohibitive task to track down and replace all these connections.

In order to facilitate changes in the fundamental components of the software, a modular message passing class based design has been designed. An abstract entity diagram of the software created can be seen in fig. 2.1

Please note that fig. 2.1 shows only a simplified overview of the software, illustrating only major components of the design. Blue components have knowledge of [Jena](#) where red components have knowledge of [JGraphX](#). Green elements are purely [Swing](#)-based and have no knowledge of either framework.

MainGUI Facilitates communication between [GUI](#) components and *DataStorage*, and is the main [GUI](#) entity. *MainGUI* is decoupled from [Jena](#) by way of *DataS-*

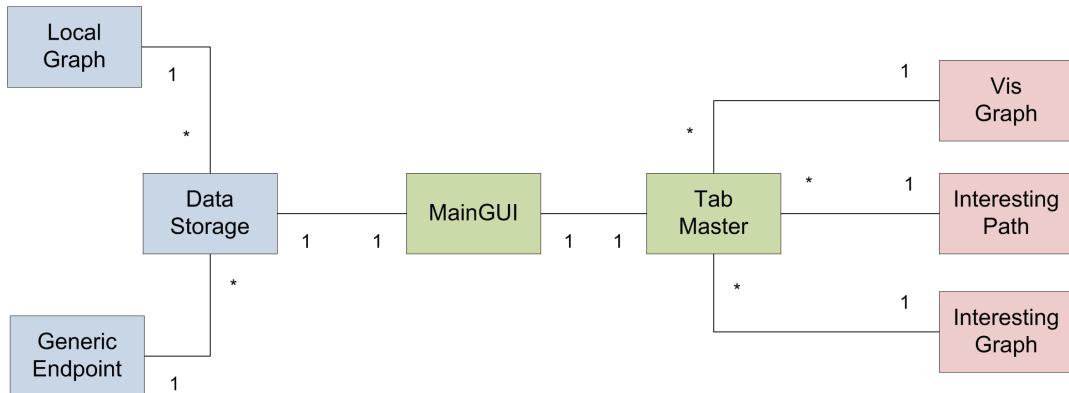


Figure 2.1: Simplified entity diagram for *GraphHelper*

torage and has no knowledge of the underlying framework used. *MainGUI* is also decoupled from [JGraphX](#) via *TabMaster*.

DataStorage An accesspoint between *MainGUI* and models and endpoints. *DataStorage* exposes a set methods for interacting with the models and endpoints through a predefined interface. *DataStorage* can have zero or more *LocalGraphs* and *GenericEndpoints*. As *DataStorage* handles some direct interaction with models, such as loading models from files, *DataStorage* is directly connected with [Jena](#). The class diagram for *DataStorage* can be found in section [E.6](#)

LocalGraph Contains one [RDF](#) graph loaded from a local file. *LocalGraph* can be interacted with using methods from the interface *DataSourceInterface*. *LocalGraph* and its interface class diagram can be found in section [E.5](#)

GenericEndpoint Contains contact information for one remote [Triplestore SPARQL](#) endpoint, such as [Open Link Virtuoso](#). *LocalGraph* and *GenericEndpoint* implements the same interface, and as such allows for the same operations, although their behaviour can be different. *GenericEndpoint* and its interface class diagram can be found in section [E.5](#)

TabMaster A custom *JTabbedPane* which controls interactions with the visualisations and the [SPARQL](#) result table.

VisGraph A wrapper around the graph, component and settings used by [JGraphX](#). *VisGraph* is only interacted with through *TabMaster* through the interface *GraphTab*. *VisGraph* and its interface class diagram can be found in section [E.2](#)

Interesting Graph Contains graph, component, settings and table used by [JGraphX](#). *Interesting Graph* is only interacted with through *TabMaster*. *Interesting Graph* and its interface class diagram can be found in section [E.2](#)

Interesting Path Consists of several classes handling graph, component and settings for visualisation, as well as a table to list paths. The class diagram for **Interesting Path** can be found in section E.2

DataStorage, *LocalGraph* and *GenericEndpoint* all use **Jena**, and are therefore considered as one *module*, in that all three would have to be rewritten in case of a change of framework. In order for the decoupled communication between *DataStorage* and *MainGUI* to remain consistent, the implementation uses strongly defined enums and custom classes for passing messages and data. *LocalGraph* and *GenericEndpoint* both implement a common interface in order to allow for future flexibility. Some triplestores already provide custom connectors for **Jena**, such as the *Virtuoso Jena Provider*[60], and it seems reasonable that these providers will provide better programmability and performance, when compared to *GenericEndpoint*.

The communication between *MainGUI* and the **JGraphX** components is much less extensive, and therefore there is only limited homogeneity in interfacing between these entities. Either of the three visualisation components could be replaced with only relative little impact on the rest of the application.

The following sections describe implementation of **Jena** and **JGraphX** in more details.

2.3 Jena Implementation

There are several ways to represent an **RDF** container in **Jena**. The most common way is to use the **Jena Model** interface, and generate the model using the static methods of *ModelFactory*. *ModelFactory* can create **RDF**, RDFS as well as other models. The *Model* interface defines a set of methods enabling the user to store nodes, properties and literals representing an **RDF** as well as a selection of methods for interacting with the model. The storage of **RDF** models can be abstracted to in-memory data structures, disk based persistent stores and inference engines. If there is a need to create custom based data-abstractions for storing triples, the **Jena Graph** interface will be more appropriate, as it has a simpler **API** than the *Model* interface, making it easier to implement custom based **RDF** stores. In most cases the default model from *ModelFactory* will be sufficient and will here after be referred to as the model.

In a model nodes can be represented as either resources, literals or blank nodes. In **Jena** blank nodes are referred to as either bNodes or anonymous node. The **Jena** interface called *Resource* represents regular **URI** resources, bNodes as well as properties, while the **Jena Literal** interface represents literals. Both interfaces share the same interface **RDFNode** which can therefore represent all four types of nodes.

As mentioned in section 1.2.1, **RDF** graphs can be broken down into triple statements, each of which contain a *subject*, *predicate* and *object*. One triple is represented in **Jena** with an instance from the *Statement* interface for named graphs. The subject in a triple can only be represented by a resource, as only blank- and **URI** nodes can

be subjects, while an object can be any type of *RDFNode*. The **Jena** *Property* class is a subclass of the **Jena** *Resource* class, and represents the predicates between the subjects and objects in the triple. The elements of a triple can be extracted from the *Statement* object with the following three methods:

- `getSubject()` will return an instance of type *Resource*
- `getObject()` will return an instance of type *RDFNode*
- `getPredicate()` will return an instance of type *Property*

Section 2.3.1 describes the functionality of the data management, and in sections 2.3.2 and 2.3.3 we shall go into more detail with how **Jena** is used in *GraphHelper*.

2.3.1 Data Management

DataStorage is the entry point for all interactions between the **GUI** and the underlying **RDF** data. *DataStorage* implements the interface *DataRepositoryInterface* for which the class diagram can be found in section E.6. All models are stored in a hash map which maps a model name to *DataSourceInterfaces*. A large proportion of *DataStorage*'s public methods contains very little logic, in that *DataStorage* forwards the request to the appropriate model, and returns the results to the caller. *DataStorage* only implements methods which act between models, or which are not directly related to any model. Listing 2.1 shows one of these method, *getQueryType*, which is not directly related to any model:

Listing 2.1: *getQueryType* determines **SPARQL** query type

```
1 public SPARQLExpressionType getQueryType(String _query) {
2     try {
3         Query query = QueryFactory.create(_query);
4         if (query.isSelectType()) {
5             return SPARQLExpressionType.SELECT;
6         } else if (query.isConstructType()) {
7             return SPARQLExpressionType.CONSTRUCT;
8         } else if (query.isAskType()) {
9             return SPARQLExpressionType.ASK;
10        } else if (query.isDescribeType()) {
11            return SPARQLExpressionType.DESCRIBE;
12        } else if (query.isUnknownType()) {
13            return SPARQLExpressionType.UNKNOWN;
14        } else {
15            return SPARQLExpressionType.ERROR;
16        }
17    } catch (Exception e) {
18        return SPARQLExpressionType.ERROR;
19    }
20 }
```

getQueryType can determine the type of a **SPARQL** query as well as determine whether the query is malformed. This method is used by the **GUI** in order to invoke the appropriate type of method on a model, as well as to stop the query process early if the expression entered is invalid.

In addition *DataStorage* handles creation, manipulation and destruction of *Data-SourceInterfaces* objects. For endpoints this is relatively simple, as only the name and **Uniform Resource Locator (URL)** need be provided. For *LocalGraph* a valid graph file is loaded using **Jena's FileManager**, as seen in listing 2.2

Listing 2.2: A file is loaded using FileManager

```

1 public ImportModel ImportModel(String _path, String
2   _modelName, FileFormat _format) {
3
4   Model model = ModelFactory.createDefaultModel();
5   InputStream in = FileManager.get().open(_path);
6   String format = ...;
7
8   if (in == null) {
9     return ImportModel.FileNotFound;
10 } else {
11   try {
12     model.read(in, null, format);
13     in.close();
14     LocalGraph newGraph = new LocalGraph(model);
15     if (this.myModels.containsKey(_modelName)) {
16       this.myModels.remove(_modelName);
17       this.myModels.put(_modelName, newGraph);
18     } else {
19       this.myModels.put(_modelName, newGraph);
20     }
21     myModels.put(_modelName, newGraph);
22   } catch (Exception e) {
23     return ImportModel.FileParseError;
24   }
25   return ImportModel.Success;
26 }
```

Listing 2.2 creates a default model and provides the *FileManager* with a file path. Lines 11-20 loads the data into the model and adds the model to *myModels*, overwriting any model of the same name already loaded.

In the following sections, more details will be provided about local file models and endpoints.

2.3.2 Local File Models

RDF graphs loaded from the various serialization formats are stored in *GraphHelper* as *LocalGraph* objects, the class diagram for which can be found in section E.5. A graph is loaded into a [Jena](#) model and stored as the private attribute *myFullModel*. In addition to the model, *LocalGraph* contains attributes such as maximum number

of nodes to be returned on visualisation calls, as well as the current center node. *LocalGraph* implements the interface *DataSourceInterface*, which dictates a large set of methods to be implemented, some of which are not relevant to *Localgraph*. Methods such as *verifyConnection* and *get/setTimeout* always return a default value.

SPARQL Support

SPARQL expressions are supported for the four query types(see section 1.2.2), by way of four methods. Listing 2.3 shows the implementation for SPARQL construct calls

Listing 2.3: Construct queries return a model upon successful completion

```

1 @Override
2 public Model performConstruct(String _query) throws
   QueryExceptionHTTP
3 {
4     Model newModel = null;
5
6     Query query = QueryFactory.create(_query);
7     QueryExecution qExe =
        QueryExecutionFactory.create(query,
          this.myFullModel);
8
9     newModel = qExe.execConstruct();
10    return newModel;
11 }
```

Lines 6-7 creates first a query object, and secondly a query plan to be executed. Said plan is executed in line 9, and if successful the resulting model is returned from the function. A model generated by construct queries are assigned a new name by *DataStorage* and is then ready for a user to work with.

Visualisation

A LocalGraph can be visualised if it already has a centre vertex, or if one is provided as part of the visualisation call. When *CenterOn* is called on a LocalGraph, it will perform a Breadth-First-Search (BrFS) going outwards from the centre node, as shown in listing 2.4

Listing 2.4: CenterOn performs BrFS

```
1 public VisualiseWrapper CenterOn() {  
2 ...  
3 while (!myQ.isEmpty())  
4     && size(subModel) < maxVertices) {  
5     RDFNode center = myQ.remove();  
6     CenterOnProcessNode(processedNodes, subModel, center,  
7                           myQ);  
8     myStatements = getStatements(subModel);  
9     result = new VisualiseWrapper(myStatements,  
10                                VisualiseStatus.Success);  
11    return result;  
12 }
```

Listing 2.4 builds a temporary model around the initial node, by processing each neighbour in an outwards fashion. Each node in the queue is processed as shown in listing 2.5

Listing 2.5: Each new node is processed in order to find new statements and frontier candidates

```

1 private void CenterOnProcessNode(...) {
2
3     StmtIterator stmts;
4     if (!_c.isLiteral()) {
5         stmts = myFullModel.listStatements(_c.asResource(),
6                                         null, (RDFNode) null);
7         while (stmts.hasNext() && size(_m) < maxVertices) {
8             Statement stmt = stmts.next();
9
10            if (!_m.contains(stmt)) {
11                _m.add(stmt);
12
13                if (!_p.contains(stmt.getObject())) {
14                    _p.add(stmt.getObject());
15                    _q.add(stmt.getObject());
16                }
17            }
18        }
19    ...
20 }
```

If the current node is not a literal, a statement iterator is created in line 5 which fetches all statements where `_c` is subject. Wildcards in [Jena](#) are denoted with `null`, and the triple patterns to look for is written as

$$<_c, *, *>$$

Lines 6-17 iterates across the statements found, and if a statement has not been processed before, it is added to the temporary model. If the object of the statement has not been processed either, it gets added to the frontier. Listing 2.5 is abridged, as the same steps are performed for statements where `_c` is the object.

2.3.3 Endpoints

The `GenericHTTPEndPoint` class represents a single remote [Triplestore](#), the class diagram of which can be found in section E.5. The [Triplestore](#) is identified by a [URL](#) attribute, which is used extensively throughout the class. As there is no local model to directly access, all interactions with the [Triplestore](#) have to be performed via SPARQL expressions. [Jena](#) supports both plain HTTP calls, as well as [Simple Object](#)

Access Protocol ([SOAP](#))-based calls. As with *LocalGraph*, *GenericHTTPEndPoint* implements the interface *DataSourceInterface* which dictates a number of methods to be implemented. The simplest method, *verifyConnection*, simply verifies that the *Triplestore* in question is available. Listing 2.6 shows the implementation

Listing 2.6: *verifyConnection* asks a *Triplestore* whether it contains anything

```

1 public boolean verifyConnection() {
2     String q = "ASK { }";
3
4     try {
5         Query myQuery = QueryFactory.create(q);
6         QueryExecution qExe =
7             QueryExecutionFactory.sparqlService(this.myURL,
8                 myQuery);
9         qExe.setTimeout(myTimeout);
10        boolean result = qExe.execAsk();
11        qExe.close();
12        return result;
13    } catch (Exception e) {
14        return false;
15    }
16 }
```

verifyConnection simply asks a *Triplestore* whether it contains any information. If it does not, or if an exception was encountered, it returns false. When working with endpoints, it is necessary to set time-outs in order to prevent remote calls that never finish. Line 7 sets the time-out to a value specified by *GenericHTTPEndPoint*, which per default is set to 30 seconds.

Visualisation

In order to create a visualisation around an initial node, *GenericHTTPEndPoint* needs to create a sub-graph in memory. Visualisation is therefore a two-step process, where a sub-graph is constructed, and then reduced to fit the maximum number of nodes specified. Listing 2.7 shows the first [SPARQL](#) expression executed

Listing 2.7: Constructs a sub-graph from all the neighbours of %s

```

1 CONSTRUCT {
2   # k = 1
3   <%s> ?p11 ?o11.
4   ?s12 ?p12 <%s>.
5 }
6 WHERE {
7   # k = 1
8   <%s> ?p11 ?o11.
9   ?s12 ?p12 <%s>.
10 }
```

The variable `%s` is the center node. If the result of listing 2.7 is less than the maximum number of nodes specified, the model is expanded up to two times with the next step of neighbours. If a time-out happens during expansion, the previously retrieved model is used. As the sub-model can be substantially larger than the maximum number of nodes, the final model is reduced using a BrFS algorithm.

Jena, Endpoints and bNodes

During development of visualisation-features for endpoints it was noted that Jena was unable to fetch neighbours of blank nodes, a feature that worked as expected for local models. After further investigation it was discovered that Jena does not use the ID assigned to a blank node by a Triplestore, but instead creates a new unique ID within the current graph. If a blank node is made a focal point, the ID sent to a Triplestore has no meaning within the Triplestores scope. This is not a flaw in Jena, but rather a design decision. The ID provided by a Triplestore to a blank node need not be consistent between SPARQL calls, as described by Jena contributor Rob V[25].

And even with CONSTRUCT queries the situation is similar, almost all RDF formats say that a blank node label is only scoped to the document. So if I have `_:id` and `_:id` in two separate requests semantically speaking I have two different blank nodes.

Regardless of the format you also have the issue that some syntaxes are quite restrictive in what characters can appear in a blank node label so even if a store does use its internal identifiers (which is rare) it will often have to escape/encode them in some way to be valid syntax. This then requires you to be aware of each endpoints escaping/encoding scheme (if it exposes identifiers at all) and how to translate it into an actual ID.

Directly addressing blank nodes located in [Triplestores](#) therefore seems problematic, and could invoke odd behaviour depending on the endpoint addressed. Blank nodes have therefore been made un-selectable for centering within *GraphHelper* until a robust method has been found for addressing blank nodes in [Triplestores](#).

2.4 JGraphX Implementation

JGraphX is used to create all visualisations for this project. The framework[44] is aimed towards modelling of mathematical graphs, flowcharts, process diagrams and more. The major classes for working with **JGraphX** are

mxGraph - The graph object from which diagrams can be created. Contains both object-components of the graph as well as styles.

mxGraphComponent - The [Swing](#) component that actually visualises a graph. This class extends *JScrollPane* and *Printable*, and contains several *EventHandler* sources.

mxConstants - Contains a large collection of constants, mostly focusing on style-options and their relevant options.

mxGraphLayout - An interface from which several automatic layout algorithms inherit.

The main interaction class is **mxGraph** as it is where the actual graph is manipulated. Vertices and edges in **mxGraph** are classified as **mxCells**. Listing 2.8 shows the insertion of two vertices and the connection of the vertices with an edge.

Listing 2.8: Creation of two vertices and an edge

```

1 Object parent = getDefaultParent();
2 getModel().beginUpdate();
3 try {
4     Object v1 = insertVertex(parent, null, "Vertex 1",
5         100, 100, 80, 30);
6     Object v2 = insertVertex(parent, null, "Vertex 2",
7         100, 100, 80, 30);
8     Object edge = insertEdge(parent, null, edgeData, v1,
9         v2);
10 } finally {
11     this.getModel().endUpdate();
12 }
```

Line 1 gets the top most component, namely the default parent. Objects will be added hierarchically to the default parent, although for truly hierarchical diagrams such as organisational diagrams elements should be added to their actual parent. Line 2 prepares the graph for bulk transaction, temporarily disabling certain event triggers for performance reasons.

Line 3-5 creates two objects and connects them with an edge. Vertices have four major properties:

ID - The second parameter passed is the ID assigned. If no ID is assigned one will be generated automatically

Value - The third parameter is a value object. The object can be a simple string, or a more complex object. If value is a complex object, the developer needs to implement methods for correctly retrieving the value that is to be shown on the vertex.

Position - Each vertex is assigned an initial X-Y position, where (0;0) is the top left corner with positive X-values moving the position to the right, and positive-Y values moving the position downwards. If the vertex is a child of another vertex, relative positions are also possible.

Dimension - A vertex is assigned a width and height.

Edges share both ID and value with vertices, but do not have positions and dimensions, but instead which vertices they are to connect. It is also possible to assign a style to a vertex or edge upon creation(see section [2.4.2](#)).

Line 8 ends the transaction and enables events.

In listing [2.8](#) the edge is given an object rather than a string as value. This object is commonly defined as an [Extensible Markup Language \(XML\)](#) document, as shown in listing [2.9](#).

Listing 2.9: Document created for multi-value storage

```

1 Document myDoc = mxDomUtils.createDocument();
2 Element edgeData = myDoc.createElement("edgeData");
3 String uri = "http://example.org/User1";
4 edgeData.setAttribute("fullURI", uri);
5 String shortURI = "User1";
6 edgeData.setAttribute("shortName", shortURI);
7 insertEdge(parent, null, edgeData, v1, v2);
```

When using non-string objects, it is necessary to override *convertValueToString* in order for labels to show properly. Listing [2.10](#)

Listing 2.10: convertValueToString() gets labels from complex values

```

1 @Override
2 public String convertValueToString(Object cell) {
3     if (cell instanceof mxCell) {
4         Object value = ((mxCell) cell).getValue();
5
6         if (value instanceof Element) {
7             Element elt = (Element) value;
8
9             if (elt.getTagName().equalsIgnoreCase("edgeData"))
10                {
11                    return elt.getAttribute("shortName");
12                }
13            }
14        }
15    return super.convertValueToString(cell);
16 }
```

As this method can be called on many different objects within a graph, robustness is important. Not all objects necessarily have to use complex value-objects at the same time.

2.4.1 Automatic Layout

While the position of vertices within a graph can be configured manually, it is often impractical to do by hand for automatically generated graphs. **JGraphX** offers several automatic methods for arranging vertices based on different requirements. A subset of these methods are shown below:

Circle - Attempts to play vertices in a circle with minimum edge distance

Organic - Organic based method where objects are pushed away from each other based on a cooling principle. A second Fast Organic version is also available

Parallel Edge Arranges vertices with more than one edge. Others remain unmoved

Stack Creates a horizontal or vertical stack layout.

As only the organic method has been employed in *GraphHelper*, this section will focus upon this methodology. The organic layout method in **JGraphX** is based on Davidson and Harzel's 1996 paper *Drawing Graphs Nicely Using Simulated Annealing*[10]. This approach considers proper laying out of a graph as an optimisation problem, where

the fitness function is a composite of several heuristics. Common components consist of:

- Even distribution of vertices
- Uniform edge-length
- Minimise edge-crossings
- Keep vertices from coming too close to edges

Only vertices with edges are affected by the organic layout algorithm. The algorithm is applied to the top most object, the parent object, as seen in listing 2.11

Listing 2.11: Fast Organic algorithm applied to a graph

```

1 mxFastOrganicLayout layout = new
    mxFastOrganicLayout(this);
2
3 this.getModel().beginUpdate();
4 try {
5     layout.execute(this.getDefaultValue());
6     } finally {
7         mxMorphing morph = new mxMorphing(myComponent);
8
9         morph.addListener(mxEvent.DONE, new
10             mxIEventListener() {
11                 @Override
12                 public void invoke(Object arg0, mxEventObject
13                     arg1) {
14                     getModel().endUpdate();
15                 }
16             });
17 }
```

Line 7 applies the layout to the graph, within a graphs-transaction. An *mxMorphing* object is also applied in order to give a visually pleasing animation to the moment of the vertices. As the *mxMorphing* is applied after the fast organic layout, the *mxMorphing* needs to end the graph-transaction post animation.

2.4.2 Customization & Styles

JGraphX facilitates the customization of canvases, vertices, edges and more. This section will focus on customisation of vertices and edges. JGraphX uses a system much reminiscent of Cascading Style Sheets, in which properties and values are defined either as a string or object, and applied to objects within a graph. Listing 2.12 shows the definition of a style as a string at the creation of a vertex.

Listing 2.12: Defining styles as strings is useful during development

```
1 String style = "shape=ellipse;strokeColor=ff000000;"  
2 Object v1 = insertVertex(parent, null, "Vertex 1", 100,  
    100, 80, 30, style);
```

This would create a black ellipse, but defining styles as strings can quickly become unwieldy and difficult to manage. When working with multiple complex styles, it is better to define a new style for the graph as seen in listing 2.13

Listing 2.13: Defining a style and adding it to the stylesheet

```
1 Hashtable<String, Object> style = new Hashtable<>();  
2 mxGraph graph = new mxGraph();  
3 mxStylesheet stylesheet = this.getStylesheet();  
4  
5 style.put(mxConstants.STYLE_SHAPE,  
    mxConstants.SHAPEROCTANGLE);  
6 style.put(mxConstants.STYLE_FILLCOLOR,  
    mxUtils.getHexColorString(Color.yellow));  
7 style.put(mxConstants.STYLE_PERIMETER,  
    mxConstants.PERIMETER_RECTANGLE);  
8 style.put(mxConstants.STYLE_STROKECOLOR,  
    mxUtils.getHexColorString(Color.black));  
9 style.put(mxConstants.STYLE_SPACING_LEFT, 5);  
10 style.put(mxConstants.STYLE_SPACING_RIGHT, 2);  
11 style.put(mxConstants.STYLE_SPACING_TOP, 3);  
12 style.put(mxConstants.STYLE_SPACING_BOTTOM, 1);  
13  
14 stylesheet.putCellStyle("Literal", style);
```

In this method, the stylesheet is fetched for the current graph, and a new style

is created using `mxConstants` which contains constants for all style options as well as some values. It is also possible to change the default style for both vertices and edges, by calling `setDefaultEdgeStyle()` and `setDefaultVertexStyle()` and passing the appropriate style as a parameter.

2.4.3 Visualisation Implementation

All implementations for visualisations are contained within the package `TabContainer`. Figure 2.2 shows a simplified class diagram for the visualisation and support classes. `TabMaster` is a customized extension of `JTabbedPane` and controls all interactions from the outside `GUI` with tabs inside it. In addition to the three types of visualisations(`VisGraph`, `InterestingGraph` and `InterestingPathsGraph`), `TabMaster` also contains one static tab which cannot be closed. This tab is used for the results of `SPARQL` queries executed else-where, and contains a `JTable` which displays the results.

`VisGraph` and `InterestingGraph` both implement the interface `GraphInterface`, which enforces communalities between the two classes so that `GraphTab` can be used as a common container. `InterestingTab` is used as a wrapper around `InterestingPathsGraph` and a `JTable`, which is listing the interesting paths found.

`GraphStyles` contains a set of static methods for the generation of `JGraphX` styles and is shared by all three `mxGraph` inheriting classes.

2.4.4 VisGraph

`VisGraph` visualises simple searches in a flash-light-like manner, where a single vertex is selected by a user when exploring a neighbourhood. The visualisation can be compared to the spotlight thrown from a flash-light, in which the view is restricted to a sphere around the centre node. The sphere area is defined based on steps from the initial node, rather than geometric distance. For a full class diagram of the `VisGraph` class, see fig. E.2. Figure 2.3 shows `VisGraph` displaying nodes around *Sir Tim Berners-Lee*.

`URI` vertices are shown as blue and literals as yellow rectangles, with blank nodes being shown as red ellipses. In order to produce a more succinct image, all `URIs` are cropped to only show the sub-string following the last forward slash or hash-tag, and literals are cropped to twenty characters. `VisGraph` takes collections of `TripleStatement`,(see fig. E.3), a custom class which contains a collection of triple statements as well as meta-information about subjects, predicates and objects. `TripleStatement` is used in order to decouple visualisation and data management, so that `VisGraph` has no awareness of `Jena` and is in no way dependent on any of `Jena`'s data structures.

Navigation & Presentation

In order to navigate around a group, the user has several options, with the primary option being to double-click an `URI`-node. This forces `VisGraph` to update to the

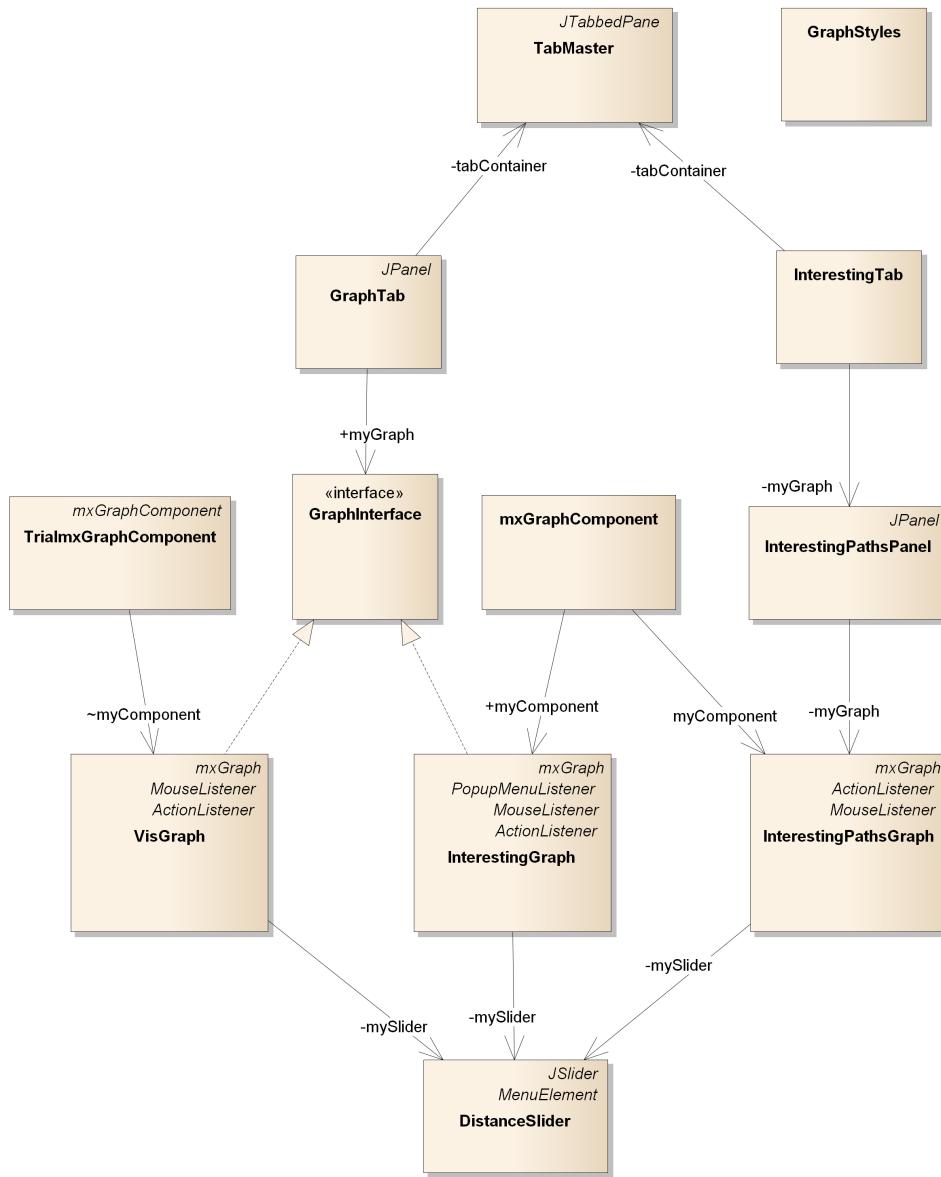


Figure 2.2: Simplified class diagram for visualisation

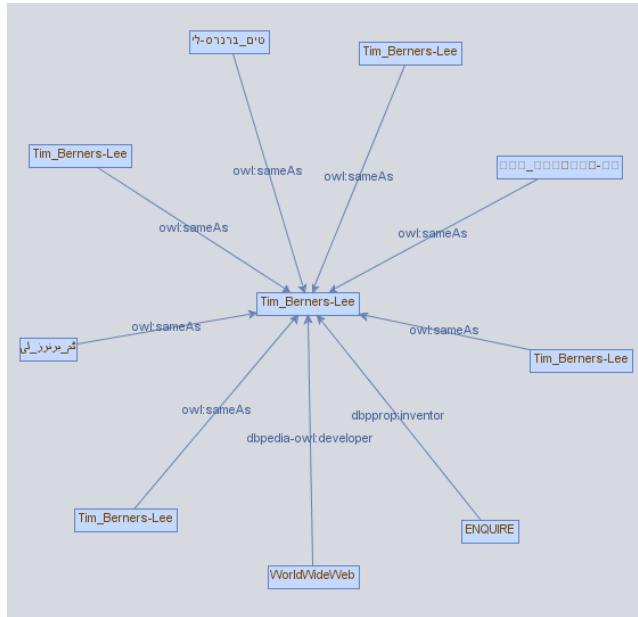


Figure 2.3: Ten neighbours of Tim Berners-Lee in DBpedia

representation with the selected vertex as the center. *VisGraph* implements a custom version of *mxGraphComponent*, which captures double clicks, the event-handler of which can be seen in listing 2.14

Listing 2.14: A custom component catches double clicks and forwards to VisGraph

```

1 public void mouseReleased(MouseEvent e) {
2     if (!e.isConsumed() && isEditEvent(e)) {
3         Object cell = getCellAt(e.getX(), e.getY(), false);
4         if (cell != null && ((mxCell) cell).isVertex()) {
5             myGraph.CenterOn((mxCell) cell);
6         }
7         e.consume();
8     }
9 }
```

The same process can be invoked by right clicking an **URI**-vertex and selecting *Center On* from the pop-up menu that appears.

Tool-tips & Menus

The user can obtain more information about a vertex in one of two ways, via tool tips or right-click menu. Tool tips are shown when a user places the cursor over an edge or vertex, at which point listing 2.15 is invoked

Listing 2.15: `getToolTipForCell()` fetches edge and vertex values

```

1 @Override
2 public String getToolTipForCell(Object cell) {
3     if (cell instanceof mxCell) {
4         mxCell c = (mxCell) cell;
5
6         if (c.isVertex()) {
7
8             return c.getId();
9         }
10        if (c.isEdge()) {
11            try {
12                Object value = c.getValue();
13                Element elt = (Element) value;
14                return elt.getAttribute("fullURI");
15            } catch (Exception e) {
16                return "Cell Element not found";
17            }
18        }
19    } else {
20        return super.getToolTipForCell(cell);
21    }
22    return null;
23 }
```

As [URI](#) vertices, blank nodes and literals all have to be unique within an RDF-graph, their respective values are used as IDs within the *VisGraph*. Edges on the other hand do not have to be unique with regard to [URIs](#), as multiple edges can have the same [URI](#). IDs for edges are therefore automatically generated, and the edge [URI](#) is stored in an [XML](#) document, which is then returned by `getToolTipForCell`.

VisGraph implements three standard [Swing](#) interfaces:

PopupMenuListener - Prototypes for events triggered when a menu appears or disappears

MouseListener - Prototypes for events triggered by mouse movement and clicks

ActionListener - Prototype for action events

VisGraph builds five custome menus to be triggered on *vertex*, *blank node*, *literal*, *edge* and *background*. When a user clicks within an area controlled by *VisGraph*, listing 2.16 is triggered

Listing 2.16: `mouseClicked()` handles mouse clicks for right-click menus

```

1  @Override
2  public void mouseClicked(MouseEvent e) {
3      if ( SwingUtilities.isRightMouseButton(e) ) {
4          JScrollPane pane = ((JScrollPane) myComponent);
5          int vertOffset =
6              pane.getVerticalScrollBar().getValue();
7          int horOffset =
8              pane.getHorizontalScrollBar().getValue();
9
10         Object[] myCells = getSelectionCells();
11         if (myCells.length == 0) {
12             myBackgroundMenu.show(myComponent,
13                 e.getX() - horOffset, e.getY() - vertOffset);
14
15         } else if (myCells.length == 1) {
16             SingleSelectedNode(myCells[0], e);
17         } else if (myCells.length > 1) {
18             MultipleSelectedNodes(myCells, e);
19     }

```

Line 3 ensures that only right-clicks are acted upon.

Lines 4-6 extracts the inherited *JScrollPane* from *mxGraphComponent* and retrieves the vertical and horizontal position. This is done in order to ensure that the position at which a menu appears is at the cursor position.

Lines 8-17 gets the number of elements current selected and triggers either a background, single element or multiple element menu. *SingleSelectedNode* further breaks this down to showing specific menus depending on the type of element clicked.

Once a menu has been spawned the user can select an item and *VisGraph* executes the desired action.

The background menu has a slider as shown in fig. 2.4 which changes the minimum distance between nodes in the current *VisGraph*. Sliding the nob has no effect until

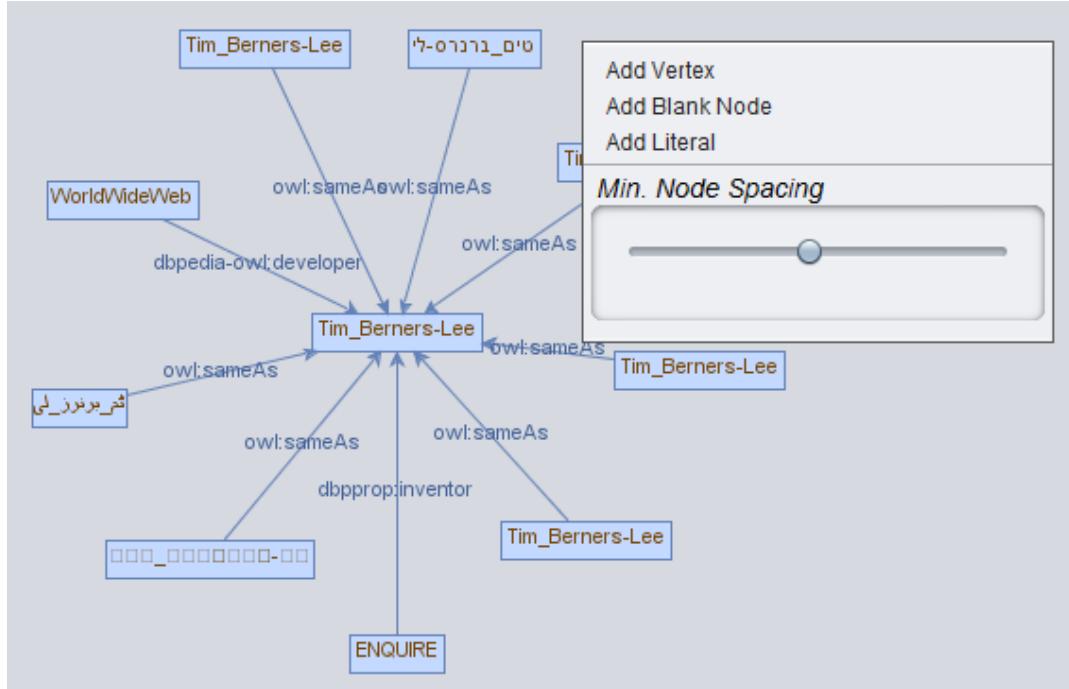


Figure 2.4: The background menu slider controls the minimum distance between nodes

the menu is closed, upon which listing 2.17 is executed

Listing 2.17: `popupMenuWillBecomeInvisible` triggers upon menus closing

```

1 @Override
2 public void popupMenuWillBecomeInvisible(PopupMenuEvent
   e) {
3   if (mySlider.valueHasChanged()) {
4     int newValue = mySlider.Update();
5     myForceConstant = (double) newValue;
6     RearrangeModel();
7   }
8 }
```

If the slider value has changed during the menus life-time, the force constant is updated and the organic layout algorithm is reapplied to the graph. This allows the user to space out denser graphs, thereby making it easier to identify and interact with individual vertices within said graph.

2.4.5 Interesting Graph

InterestingGraph, (for class diagram see section E.2), shares a large part of it's implementation with *VisGraph*, invoking features such as organic layout and menus (with fewer options). As *InterestingGraph* needs additional information in order to present a more abstract overview when compared to *VisGraph*, *InterestingGraph* uses *InterestingTriple* instead of *TripleStatement* as a data container.

In order to provide a better overview of complex graphs, *InterestingGraph* uses two distinct styles, namely *withMatch* and *withoutMatch* from *GraphStyles*, in order to colour vertices with a positive score green and red for vertices with a score of zero, as seen in fig. 2.5.

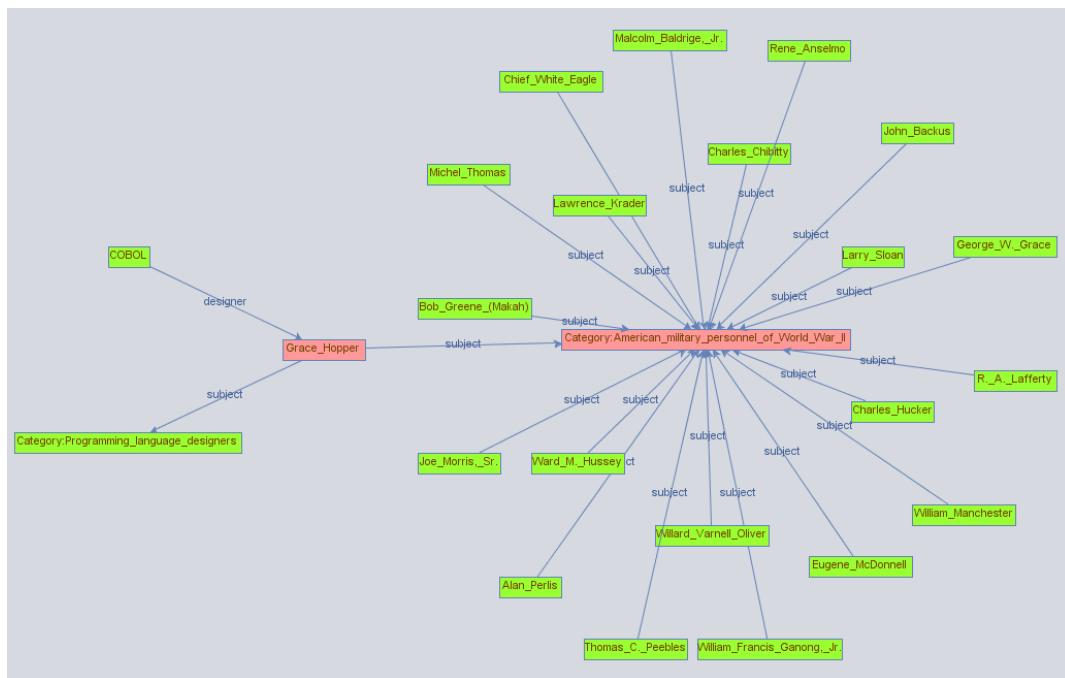


Figure 2.5: Search on Grace Hopper looking for terms: *COBOL*, *Programming Language*

In order to handle vertices with more than one predicate connecting them, *InterestingGraph* implements abstract edges. An edge can be rendered with or without arrows depending on *InterestingTriple.predDir*

source2Target - One or more predicates going from source to target. Edge will show direction

target2Source - One or more predicates going from target to source. Edge will show direction

uniDirection - Two or more predicates connecting in opposite direction. Edge will not show direction. A custom style is applied to these edges to prevent showing of arrow ends.

InterestingGraph initially only shows [URI](#)-vertices in order to keep a simple overview. The user can click individual vertices and select *Get Literals* from the right click menu. Literals for the vertex is fetched from the data source(local models extract the literals directly, where as remote endpoints executes a [SPARQL](#)-expression), attached to the vertex and the graph rearranged with the layout.

2.4.6 Interesting Paths

Interesting Paths differs from *VisGraph* and *InterestingGraphs* both graphically and in features. The visualisation of interesting paths both lists the paths in a table, as well as showing their graphical representation, as seen in fig. 2.6 The table and visu-

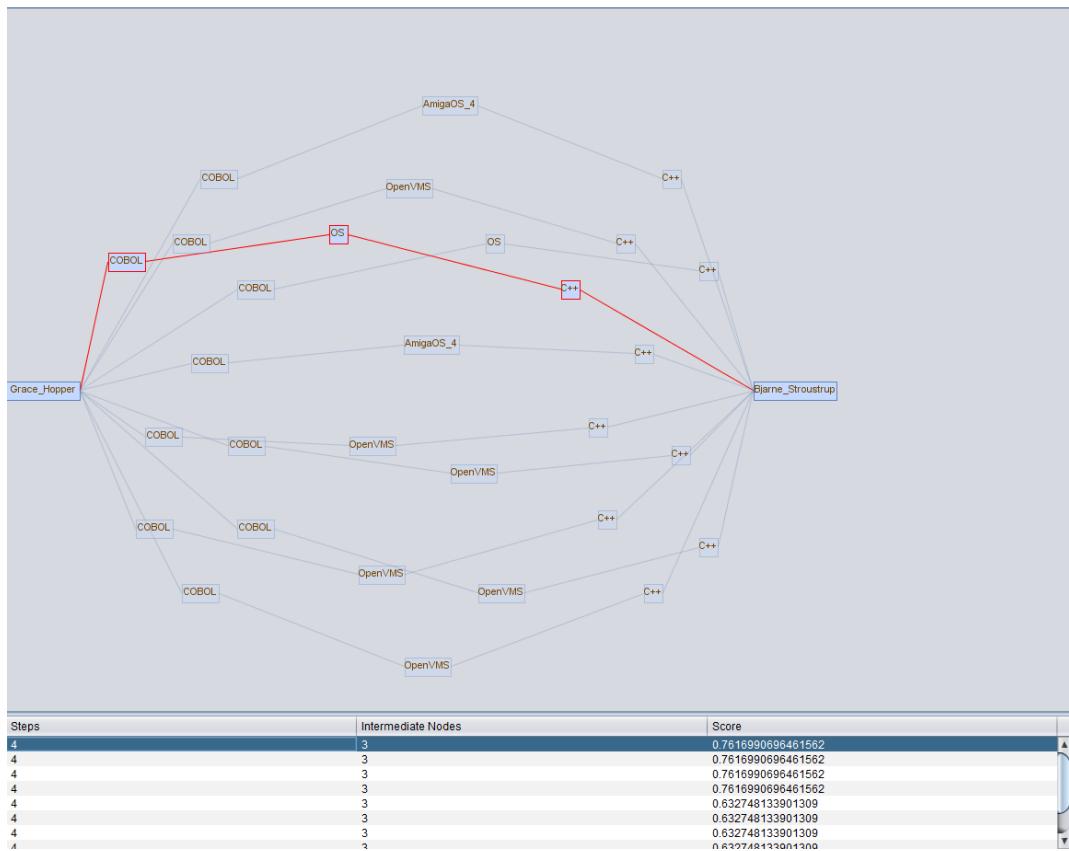


Figure 2.6: Visualisation of ten paths with one path highlighted in red

alisation work in conjunction in that a path selected in the table will be highlighted in the visualisation. Listing 2.18 shows the custom modifications implemented in *InterestingPathsPanel* to achieve this.

Listing 2.18: Custom events and settings for implemented table

```

1 myTableModel = new DefaultTableModel(
2     new Object[]{"Steps", "Intermediate Nodes", "Score"},
3     0) {
4     @Override
5     public boolean isCellEditable(int row, int column) {
6         return false;
7     };
8
9 myTable = new JTable(myTableModel);
10 myTable.getSelectionModel()
11     .addListSelectionListener(new ListSelectionListener() {
12
13     @Override
14     public void valueChanged(ListSelectionEvent e) {
15         int row = myTable.getSelectedRow();
16         if(row > -1) {
17             myGraph.setPathSelected(row);
18         }
19     }
20 });

```

Lines 1-7 makes the table uneditable, where lines 9-20 adds a listener to the table which notifies the *InterestingPathsGraph* class with the index of the newly path selected.

Path Layout

When a path is selected all entities that are not part of the selected path are greyed-out, and entities in the path are high-lighted with a red border. Listing 2.19

Listing 2.19: Selection of a path requires the application of custom styles to all elements in the graph

```

1 private void getPathObjects(int _idx) {
2     Object[] pathObjects = new
3         Object[myPaths[_idx].myPath.size()];
4
5     Object[] edges = getChildEdges(getDefaultParent());
6     Object[] pathEdges = new
7         Object[myPaths[_idx].Steps()];
8     int ptr = 0;
9
10    for (Object e : edges) {
11        if (memberOfPath(e, _idx)) {
12            pathEdges[ptr] = e;
13            ptr++;
14        }
15    }
16
17    Object[] vertex =
18        getChildVertices(getDefaultParent());
19    Object[] pathVertices = new
20        Object[myPaths[_idx].IntermediateNodes()];
21    ptr = 0;
22
23    for (Object v : vertex) {
24        if (memberOfPath(v, _idx) && !(startObj.equals(v)
25            || endObj.equals(v))) {
26            pathVertices[ptr] = v;
27            ptr++;
28        }
29    }
30    Object parent = this.getDefaultParent();
31    this.getModel().beginUpdate();
32    try {
33        setCellStyle(edgeFade, edges);
34        setCellStyle(edgeSelect, pathEdges);
35        setCellStyle(vertexFade, vertex);
36        setCellStyle(vertexSelect, pathVertices);
37        setCellStyle(this.vertex, new Object[]{startObj,
38            endObj});
39        setCellStyles(mxConstants.STYLE_MOVABLE, "false",
40            new Object[]{startObj, endObj});
41    } finally {
42        this.getModel().endUpdate();
43        refresh();
44    }
45}

```

Lines 2-24 iterates across all edges and vertices in the graph, in search of those that are members of the selected path. Lines 25-37 applies the faded style to all edges and vertices, and then un-fades those elements that are part of the selected path.

Layout & Paths

Lines 32-33 in listing 2.19 configures the start and end node as unmoveable. This is done in order to prevent the automatic layout from moving said nodes. *InterestingPaths* places the start and end node at opposite ends of the canvas, locks them, and then places all other nodes in the middle between them. Finally an organic layout is applied which pushes the non-locked elements away from each other, thereby separating out individual paths from each other. As with *VisGraph* the minimum distance between nodes can be changed using a menu slider.

2.5 Algorithms

2.5.1 Extracting Sub-Graphs

This section details the development process for sub-graph extraction used by [Interesting Paths](#) and [Interesting Graphs](#), as well as the issues that provoked changes in methodology. [Triplestores](#) can contain vast amounts of information, and in order to provide reliable service they often impose limitations upon their external [SPARQL](#) fronts. The main three restrictions discovered during the development phase are:

Time-outs: There are two types of time-out that can occur during retrieval of data namely client-side and server-side. Client-side time-outs can be configured for [Jena SPARQL](#) endpoint requests, where as server-side time-outs are defined by the endpoint. As server-side time-outs are outside the sphere of control for the application, the only possible to avoiding this type of restriction is to ensure that [SPARQL](#) expressions sent to an endpoint requires minimal processor time. Therefore sending large [SPARQL](#) expressions, forcing the endpoint to perform laborious filtering or merging tasks, are to be avoided when possible.

SPARQL-length: Some endpoints restrict the length of queries, usually by length of string. This commonly occur when an expression uses the *IN* or *VALUES* operators containing several long [URIs](#) to be matched in some way. The proper way to avoiding this restriction, is to break large expressions up into shorter queries.

Result set size: [SPARQL](#) 1.1 allows queries to set *LIMIT* and *OFFSET* thereby permitting queries to be broken up into sections. [Triplestores](#) can present limitations both on the maximum page size, as well as the maximum size of the total result set. The total result set limitation is the more stringent, as it can make it near impossible to get all connections, for a highly connected vertex.

In addition to these hard limitation there are softer restrictions that arise from communicating over a public network like the internet. The first implementation

attempted for extracting sub-graphs from a remote triple store, in this case [DBpedia](#) used a simple FIFO-queue to keep track of the frontier and performed a single [SPARQL](#) call for each [URI](#) in the frontier. In addition a set was maintained of vertices already visited. The [StringTemplate](#) for the query executed can be seen in listing 2.20

Listing 2.20: StringTemplate for extracting in and outbound connections of a vertex

```

1 INANDOUT(URI, filters) ::==
2 <<
3 SELECT (\<<URI>\> AS ?input) ?p1 ?p2 ?o ?s WHERE
4 {
5   {
6     \<<URI>\> ?p1 ?o
7     FILTER(!isBlank(?o)) .
8     FILTER(!isLiteral(?o)) .
9
10    <if(filters)>
11    <filters: {f|
12      FILTER(!strStarts(STR(?o), "<f>")) .
13      FILTER(!strStarts(STR(?p1), "<f>")) . }>
14    <endif>
15  }
16 UNION
17 {
18   ?s ?p2 \<<URI>\> .
19   FILTER(!isBlank(?s)) .
20   FILTER(!isLiteral(?s)) .
21
22   <if(filters)>
23   <filters: {f|
24     FILTER(!strStarts(STR(?s), "<f>")) .
25     FILTER(!strStarts(STR(?p2), "<f>")) . }>
26   <endif>
27 }
28 }
29 >>
```

Listing 2.20 selects all triples with the [URI](#) either as subject or object, applies filtering, and then merges the two sub-queries into one. A number of worker threads would then perform one query each and return any new [URIs](#) discovered to the frontier, as well as storing all triples retrieved in a [Jena](#) Model. This approach was

indeed capable of building sub-graphs around a vertex, but suffered from two major disadvantages:

Performance: Initially each query executed would require ~2000ms to complete. It should be noted that time measurements on remote endpoints can be only crude approximations and will depend on the endpoint, client-side hardware, internet connection etc. By restructuring the **SPARQL** query to using *SELECT* instead of *CONSTRUCT*, the retrieval time was reduced to ~600ms per call. But even 600ms per query is problematic, when it is not uncommon for vertices to have neighbourhoods on a scale of 100.000 at even a distance of two or three from the initial vertex. This would mean extracting a sub-graph of distance 2-3 would require ~16 hours. This can obviously be brought down by deploying more than one worker thread, but even here there are limitations. At a worker thread count of more than six, **DBpedia** would begin returning HTTP errors, commonly *503 - Service unavailable* and *500 - Internal server error*, which would require the resubmission of the query, thereby requiring additional time to complete the request.

Memory-usage: Keeping all retrieved triples within a **Jena** Model in memory is expensive, as some sub-graphs taking up 1G of memory or more. A single monolithic frontier and visited vertices system is also expensive, as the list of visited vertices can grow massive.

Based on the experiences from the first implementation, a new algorithm design was proposed, with the two major changes being:

Aggregated SPARQL Queries: In order to improve performance during data collection, the algorithm should fetch data for multiple **URIs** during each **SPARQL** call.

Data Offloading: The algorithm should not maintain a full model of the collected triples in memory, but instead offload to a disk-based system. The algorithm should also minimize how much data needs to be maintained within the frontier and visited vertices data structures.

Parallel Bucket-Based Breadth First Search

Parallel Bucket-Based Breadth First Search (**PBBBFS**) is the algorithm devised by the research group which incorporates these improvements. The class diagram for **PBBBFS** can be seen in fig. 2.7(for the full class diagram, see section E.1). *MetaWorkUnit* and *DataWorkUnit* extend *WorkUnit* and contain everything necessary in order to process the specific unit, such as the generation of appropriate **SPARQL** expressions and storage of partial frontier results. *MetaWorker* and *DataWorker* both extend the standard **Java** thread class, and implement their own logic for processing work units. The worker threads each retrieve and deliver work units from the *DataManager*, which ensures thread-safety by synchronizing input/output methods.

The *DataManager* contains a queue of **URIs** for *MetaWorkUnits* as well as a dual-queue structure for frontier and visited vertices. The dual-queue consists of

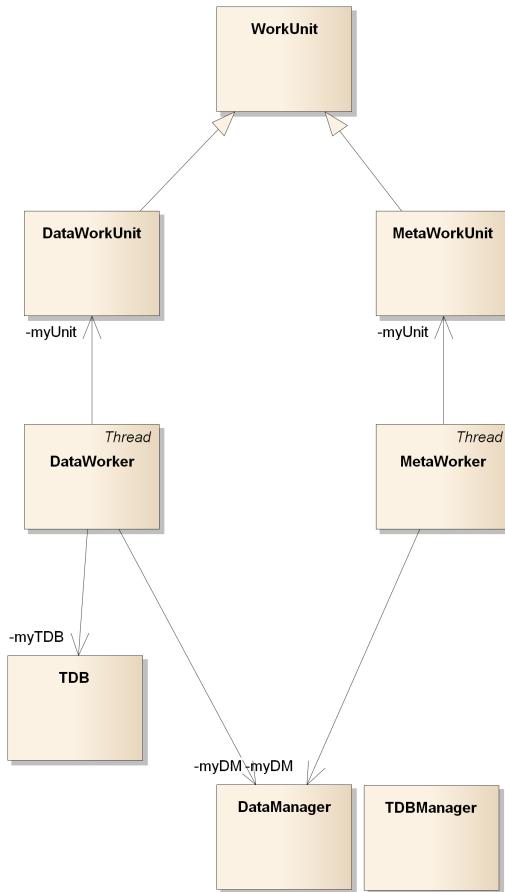


Figure 2.7: Class diagram for **PBBBFS** algorithm

a *current frontier* and a *next-step frontier*[43]. In order to prevent the algorithm from exploring already explored territory, there must be no intersection between the current and next-step frontier. Once the current frontier has been fully explored, the next-step frontier becomes the new current frontier as is seen in step two in fig. 2.8. This method of frontier-managing means that **PBBBFS** need only keep the current frontier in memory, instead of a list of all visited vertices. fig. 2.8 illustrates this method, where the red vertices are in the current frontier, and the green vertices are being added to the next-step frontier. Instead of maintaining an in-memory model **PBBBFS** exports triples to a **TDB**(for an introduction to TDB, see chapter B)

Process & Implementation

PBBBFS is a two step iterative multi-threaded process, illustrated by the flowchart in fig. 2.9

PBBBFS will iterate until the model distance, that is the number of steps from the initial vertex out to the frontier, is equal to the target distance. Each iteration consists of three processes executed in sequence:

Exploration Process: In order to be able to aggregate **URIs** from the frontier into groups that can be extracted, the algorithm needs to know the result size for each **URI** in the current frontier. The exploration process gathers this meta information by aggregates **URIs** in the current frontier into groups and creates **SPARQL** expressions using listing 2.21

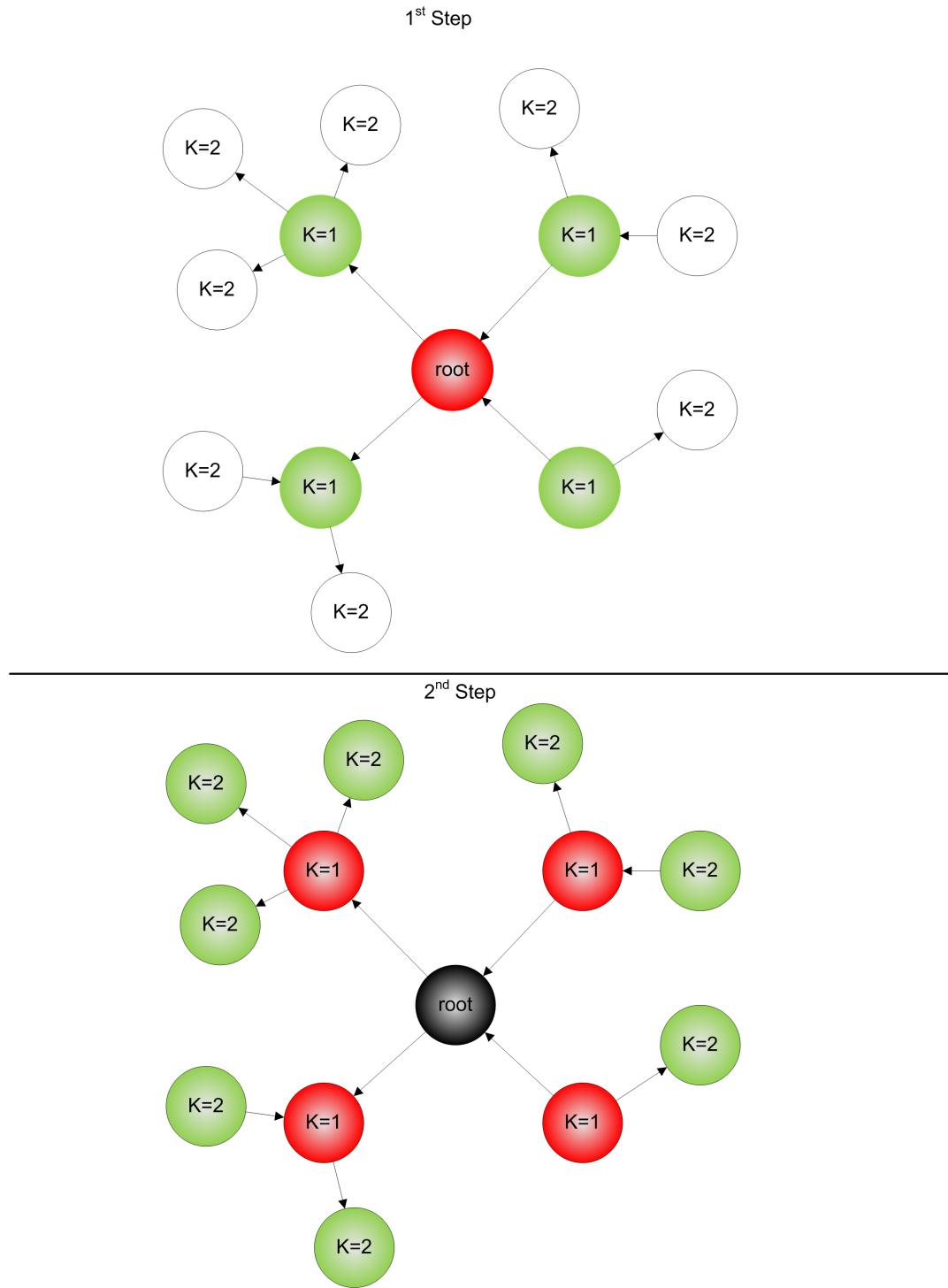
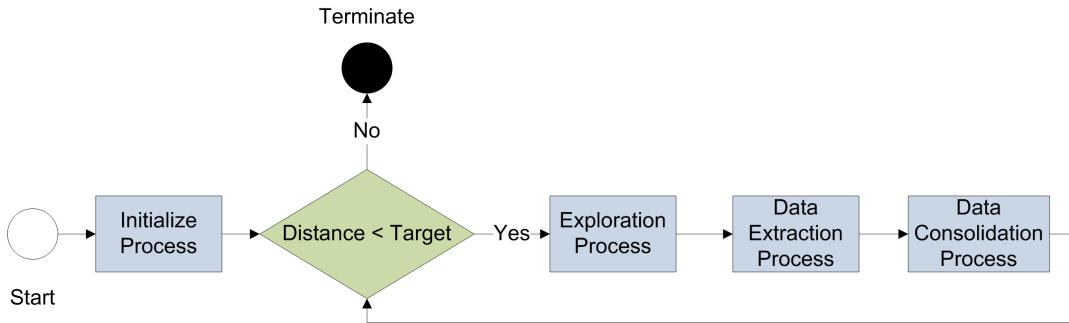


Figure 2.8: PBBBFS uses a concentric ring-based approach to frontier-management

**Figure 2.9:** Flowchart diagram for PBBBFS**Listing 2.21:** StringTemplate for retrieving meta data

```

1 COUNT_TRIPLES(URIs, filters) ::=
2 <<
3 SELECT ?known (COUNT(?known) AS ?count) WHERE
4 {
5   VALUES ?known {<URIs: {uri|\<<uri>\>}>} .
6   {
7     ?known ?p ?o .
8     FILTER(!isLiteral(?o)) .
9     FILTER(!isBlank(?o)) .
10    <if(filters)>
11    <filters: {f|
12      FILTER(!strStarts(STR(?o), "<f>")) .
13      FILTER(!strStarts(STR(?p), "<f>")) .} >
14    <endif>
15  }
16  UNION
17  {
18    ?s ?p ?known .
19    FILTER(!isLiteral(?s)) .
20    FILTER(!isBlank(?s)) .
21    <if(filters)>
22    <filters: {f|
23      FILTER(!strStarts(STR(?s), "<f>")) .
24      FILTER(!strStarts(STR(?p), "<f>")) .} >
25    <endif>
26  }
27 } GROUP BY ?known
28 >>
  
```

listing 2.21 selects all triples involving `?known` and groups them, returning the aggregated results as the output. `VALUES` contains a list of `URI` for which to determine number of triples. The entire expression is kept below 4000 characters in length, an empirically determined value, in order to avoid the `SPARQL` length restriction.

The Exploration Process can be seen in fig. 2.10 The process spawns a number

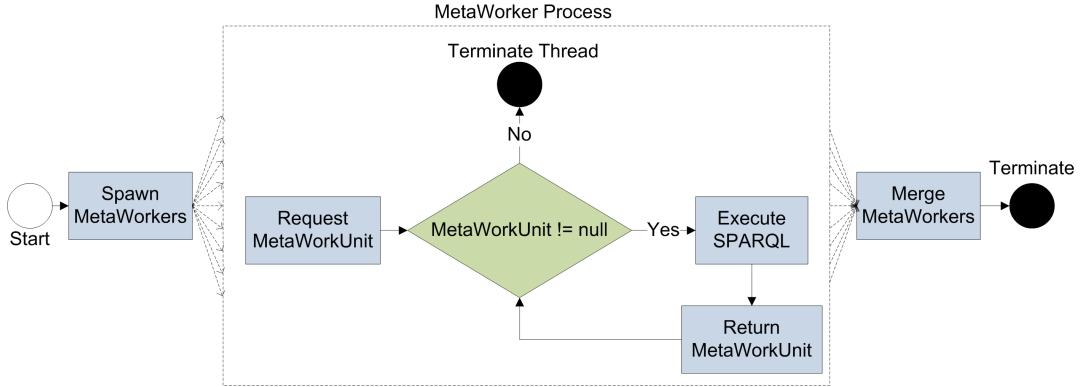


Figure 2.10: Process diagram for Exploration Process

of *MetaWorker* threads equal to the number of cores available to the `JVM`. The *MetaWorkers* will continue to request and process work units, until it is handed a null package, upon which the thread will terminate. Once all threads have finished, the sub-process ends.

Data Extraction Process: Using the meta-information gathered in the previous step, this process groups `URIs` into *DataWorkUnits* and hands these off to *DataWorkers* as illustrated in fig. 2.11 *DataWorkUnits* come in two types; normal,

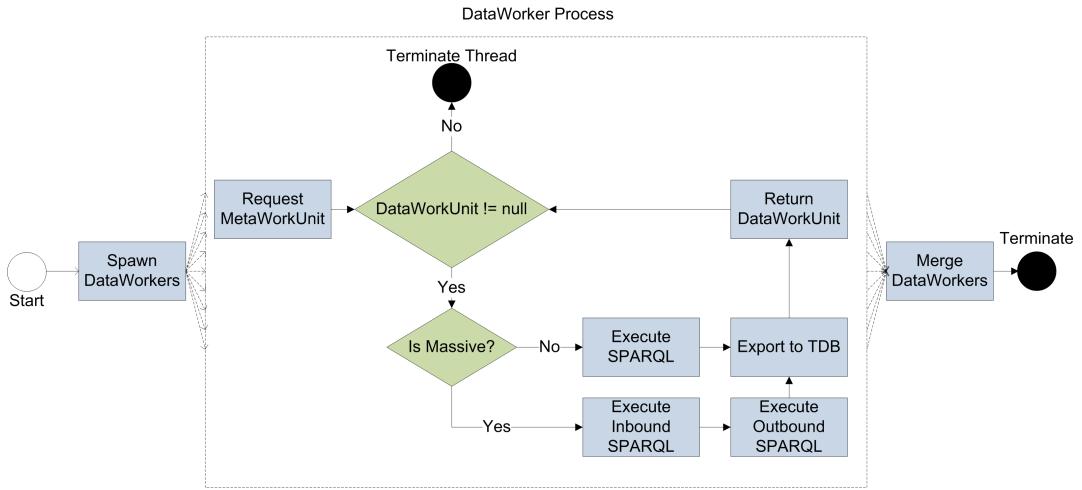


Figure 2.11: Process diagram for the Data Extraction Process

defined as those with a result set of 10.000 or less, and massive for those with a result set larger than 10.000. Normal *DataWorkUnits* are processed with a single **SPARQL** call that collects both in- and outgoing triples. Massive *DataWorkUnits* are split into in- and outgoing triples, and then each subset is explored up to a depth of 40.000 results. While this method does not ensure a full collection, it does retrieves the largest set possible from a remote endpoint given the limitations mentioned earlier. Once the triples have been collected, they are handed off to a **TDB** triple store(for information about TDB, see chapter B), and the potential new frontier vertices are handed over to the *DataManager*.

2.5.2 Interesting Graph

Finding related material in an **RDF** graph can be a daunting task. It is not uncommon for vertices to have neighbours numbering in the hundreds or thousands. While interfaces such as **SPARQL** can be used to explore neighbourhoods, the result sizes can quickly become unmanageable for human users. To assist users in exploring **RDF** graphs, the research group has developed **Interesting Graph**, an **A***(pronounced 'A star')[32]-like search algorithm.

Algorithmic Behaviour

A* is an approximative graph search algorithm that relies on heuristics in order to guide the search. It has a wide range of applications, from efficient path finding in video games to general graph searches. Instead of performing a greedy-search to explore a neighbourhood, **A*** prioritizes exploration with a heuristic measure. **A*** implements a priority queue as a frontier, in which the best candidate nodes are ordered based on fitness. Candidates are commonly scored based on distance and a heuristic

$$f(n) = g(n) + h(n) \quad (2.1)$$

Where

$f(n)$ is the total fitness score for node n

$g(n)$ is the distance from the initial node for node n

$h(n)$ the heuristic score for node n

The heuristic depends on the scenario in which **A*** is deployed. For finding the shortest distance between two nodes, $h(n)$ might be the distance from n to the target. **A*** can therefore behave in two different ways

BrFS As $h(n)$ approaches 0, **A*** will explore the closest neighbours first

Best-First-Search (BeFS) As $h(n)$ increases proportionally to $g(n)$, **A*** will explore promising neighbours more aggressively

Interesting Graph is not a full **A*** implementation in that it does not have a *target node* to search for. Instead the heuristic focuses on estimating the interestingness of nodes. First we define a few terms:

- $j \rightarrow$ The j -th node
- $L(j) \rightarrow$ Function returning a **bag-of-words** set of literals for node j
- $T \rightarrow$ **bag-of-words** set for search terms
- $H_j \rightarrow$ The number of search term hits for node j
- $ms_j \rightarrow$ Weighted match score for node j
- $s_j \rightarrow$ Score for node j
- $\beta \rightarrow$ User specified weight

The number of term matches is calculated using eq. (2.2)

$$H_j = T \cap L(j) \quad (2.2)$$

and the score for the node is calculated using eq. (2.3)

$$ms_j = \beta \cdot \frac{\log(1 + H_j)}{\log(1 + |T|)} \quad (2.3)$$

Figure 2.12 shows a plot of ten match scores with term set cardinality ranging from one to ten, with $\beta = 1$. Equation (2.3) assigns scores in a non-linear fashion, assigning

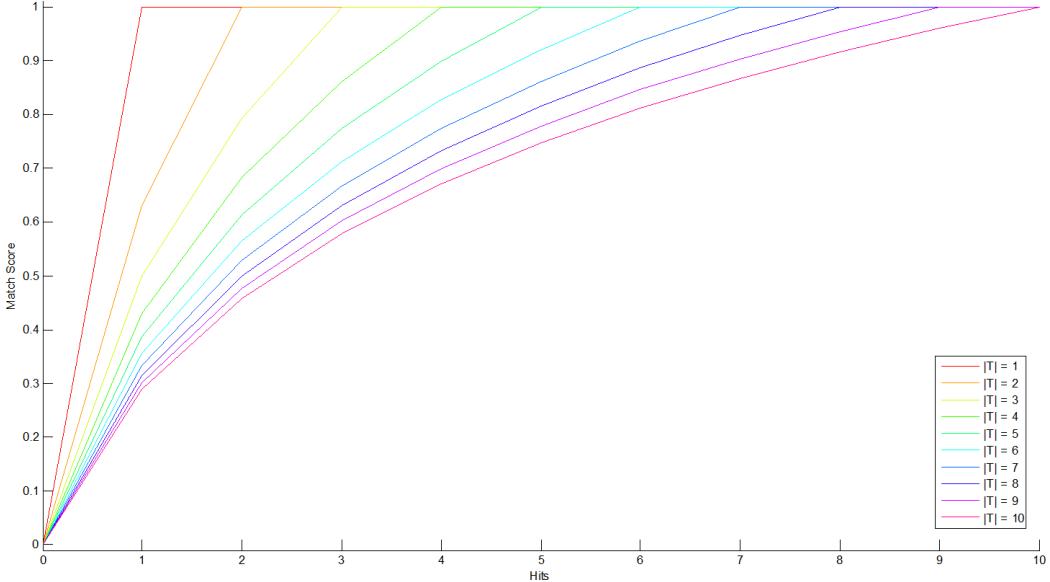


Figure 2.12: Equation (2.3) with $|T| \in [1; 10]$ and $\beta = 1$

relatively high scores to low match counts. This is done in order to ensure that

[Interesting Graph](#) doesn't ignore single term matches. β allows a user to adjust the aggressiveness with which [Interesting Graph](#) should pursue interesting nodes. For $\beta = 0$ [Interesting Graph](#) acts as an [BrFS](#), and increasingly higher values of β forces [Interesting Graph](#) to search in a more [BeFS](#) manner.

Implementation

For endpoints [Interesting Graph](#) uses [TDB](#)(for details on [TDB](#), see chapter [B](#)), and as such the implementation for local graph files and endpoints are much the same and will be treated as one. The process for [Interesting Graph](#) is illustrated in fig. 2.13

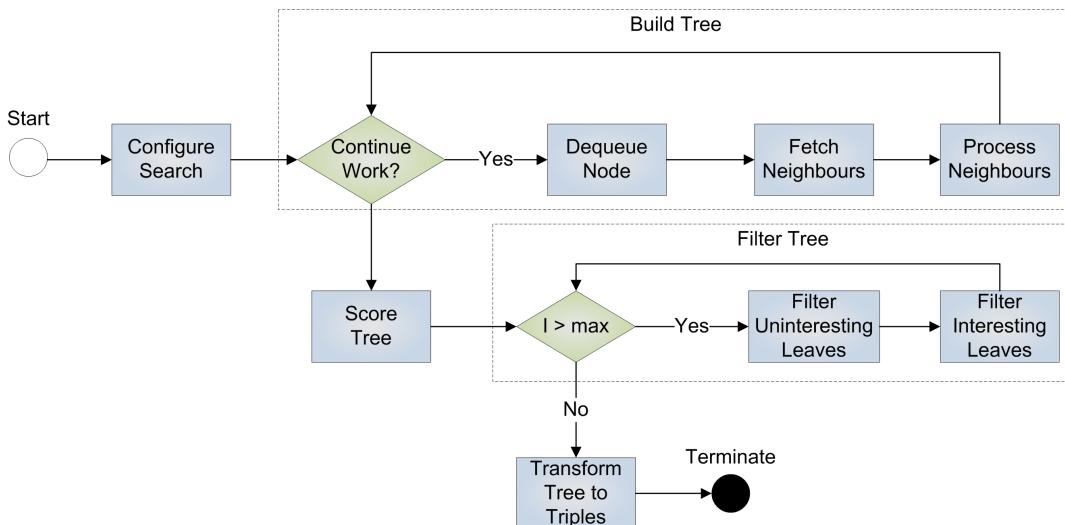


Figure 2.13: Process diagram for [Interesting Graph](#)

Build Tree: [Interesting Graph](#) builds a custom [N-ary Tree](#) [RDFTree](#) of [RDFTreeNodes](#)(see section [E.4](#) for class diagrams) branching out from the initial node. A frontier and list is used to keep track of which nodes to expand, and which already have been processed. [RDFTree](#) keeps track of its own depth, which is used to track distance from the initial node. The tree will be expanded until either the maximum distance has been reached, or the frontier is empty. Once the tree has been built, the process moves on to scoring.

Score Tree: The tree is not scoring during construction, as this would entail extracting literals for each node sequentially. During development of [PBBBFS](#)(see section [2.5.1](#)), clear performance advantages was achieved by bundling [URIs](#) into sets of [SPARQL](#) expressions, rather than performing individual calls. [Interesting Graph](#) therefore constructs a set of [SPARQL](#) queries for extraction of literals which are stores in a one-to-many collection using the [Google Guava](#) v1.7 library. The algorithm now has sufficient information to select a subset of the tree to present to the user.

Filter Tree: The process of reducing the *RDFTree* follows two rules:

Connectivity - All nodes should be connected and *floating* unconnected nodes are not allowed

Priority - Removal of nodes should favour *uninteresting* followed by *least interesting* nodes

To accommodate these restrictions, a filtering strategy has been developed. Figure 2.14 shows an example of an *RDFTree*. The *Connectivity* rule allows for the

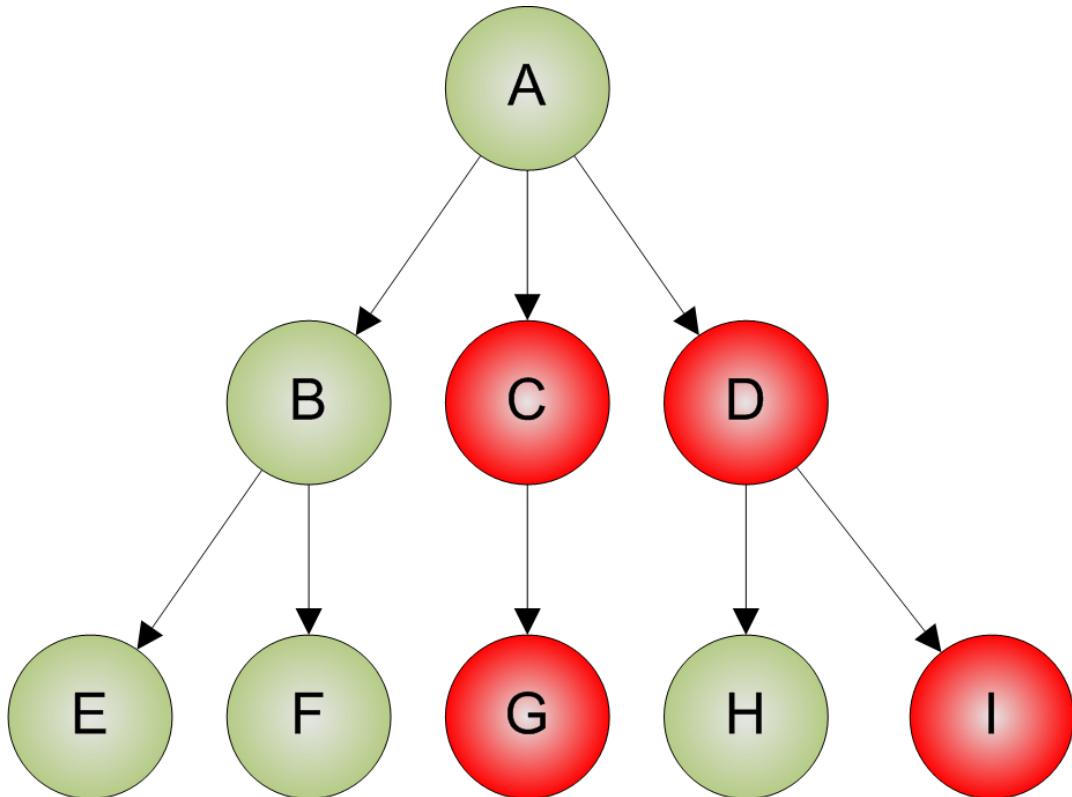


Figure 2.14: Example of *RDFTree* with interesting(green) and uninteresting(red) nodes

removal of nodes I, C and G but not D as this would leave H hanging. If I, C and G were removed, and a limit of four nodes was introduced, it would be necessary to make a choice between removing E, F and H. B is not a candidate for removal, as this would leave E and F hanging.

Filtering is implemented as the class *RecursiveInterestingPruning*, which filters *RDFTrees* as an iterative two-step process. Uninteresting nodes are filtered as shown in listing 2.22

Listing 2.22: `pruneUninterestingLeaves` removes uninteresting nodes without children

```

1 private void pruneUninterestingLeaves() {
2     List<RDFTreeNode> list = new ArrayList<>();
3
4     for (RDFTreeNode tn : myTree.getNodes()) {
5         if (!tn.hasChildren() && tn.getMyMatchScore() == 0.0)
6             {
7                 list.add(tn);
8             }
9         for(RDFTreeNode tn : list) {
10            myTree.removeNode(tn);
11        }
12    }

```

PruneUninterestingLeaves iterates over the set of nodes in the tree, generating a list of all nodes which have a score of 0.0 and no children. The nodes are removed from the tree afterwards. The second step, *pruneInterestingLeaves* is shown in listing 2.23

Listing 2.23: `pruneInterestingLeaves` removes interesting nodes from leaves

```

1 private void pruneInterestingLeaves(RDFTreeNode root) {
2     PriorityQueue<RDFTreeNode> myLeafNodes = new
3                     PriorityQueue<>(50, Collections.reverseOrder());
4
5     for (RDFTreeNode tn : myTree.getNodes()) {
6         if (tn.getMyChildren() == null ||
7             tn.getMyChildren().isEmpty()) {
8             myLeafNodes.add(tn);
9         }
10    }
11
12    while (!myLeafNodes.isEmpty() && maxInter < totalInter)
13    {
14        myTree.removeNode(myLeafNodes.remove());
15        totalInter--;
16    }
17}

```

PruneInterestingLeaves constructs a priority queue of interesting leaf nodes, and removes the least interesting node iteratively. Once the tree has been filtered, it is transformed into triples.

Transform Tree to Triples: *RDFTree* does not maintain predicate relations between nodes, and therefore in order to visualise the resulting graph, these relations have to be rediscovered. As any two nodes can have several predicates connecting them, the process finds all relevant connections. The final result is returned as an array of *InterestingTriples*(for class diagram see section E.3)

2.5.3 Interesting Paths

Where *Interesting Graph*(section 2.5.2) searches in an area around a given vertex, *Interesting Paths* searches for note-worthy paths between two vertices, given a set of restrictions. *Interesting Paths* is an extension of work done by Heim et al.[33] in the article *RelFinder: Revealing Relationships in RDF Knowledge Bases*. Heim et al. described a system for finding paths between two vertices in any given *Triplestore* using *SPARQL* expressions.

A valid path is defined as any set of triples leading from A to B, with vertices passed through called *intermediate nodes*. Certain limitations are imposed on valid paths in order to keep paths understandable to users. These limitations are:

Loops: As the number of intermediate nodes increases, the number of trivial solutions of looping through the same vertices, increases. This unnecessarily increases the number of paths that have to be processed and adds little value to the result. In order to prevent this, paths are not allowed to have repeating vertices.

Changes of direction: Paths with several changes of direction, can be difficult for users to interpret. For performance and usability reasons, the maximum number of changes in directions allowed is set to one(1). This makes for four types of connections, illustrated in fig. 2.15

Where c is an previously unknown vertex. These four types are informally named *direct path*, *shared property path* and *shared resource path*.

The result set for paths of even modest length can be quite large, numbering in the thousands, well above what any end user can realistically analyse[18]. The research group proposes to sort these paths based on interestingness, using a *bag-of-words* approach. The *bag-of-words* of search terms is provided by the user, and is used to give scores to vertices in paths found, based on the literals attached to the intermediate nodes. A simplified diagram of the process can be seen in fig. 2.16. The following sections takes a closer look at each of the four steps in the process. For experimental results of the algorithm see section 3.3

User Input

Interesting Paths takes six parameters:

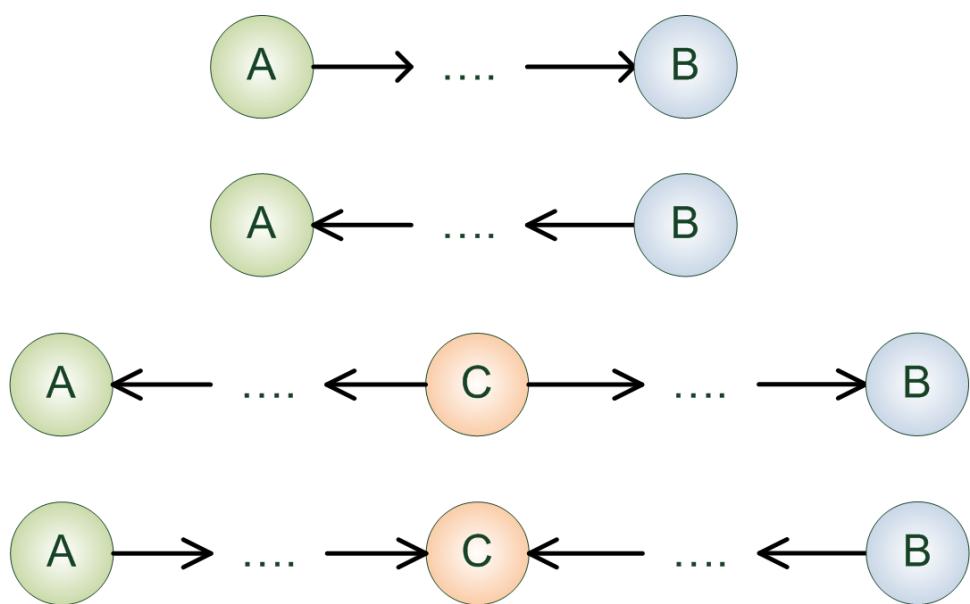


Figure 2.15: Four types of paths are permitted

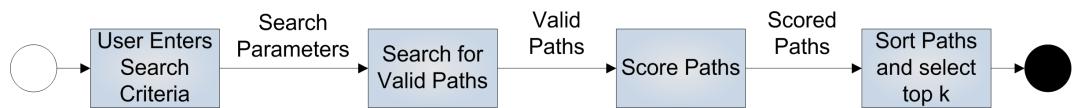


Figure 2.16: Simplified process diagram for Interesting Paths

Source/Target URI for the source and target vertex. In order to assist users in selecting appropriate URIs, a disambiguation feature has been implemented as described in chapter D

Min/Max Intermediate Nodes Minimum and maximum number of intermediate nodes between source and target. The lower bound allowed is one(1) in order to exclude the trivial solution of a direct connection

Language Language restriction for literals. Some RDF graphs assume a default language, which literals will not be tagged with. Language is therefore optional

Filters URIs for predicates and vertices which are to be ignored. A set of *default* URIs are automatically provided, filtering OWL, RDF and RDFS. Filters are optional

Search Terms Terms which describe the topic of interest to the user

Max # of Paths The maximum number of paths returned by the algorithm

Once a proper set of parameters has been provided, the next step of the process is executed.

Searching for Paths

Paths are found using a set of SPARQL expressions generated using StringTemplates. Before going into details about how paths are retrieved, the research group would like to discuss property paths. SPARQL v1.1 brought a new feature called property paths, which adds support for multi-step triple patterns. Listing 2.24 shows a simple example of a concatenated property path

Listing 2.24: Example of SPARQL property path and explicit path

```

1 ...
2 {
3   ?person rdf:type/rdf: foaf:Person .
4 }
5 {
6   ?person rdf:type ?x .
7   ?x rdf:type foaf:Person .
8 }
```

Line 2 is equivalent to lines 5-6. Property paths support a number of logical operators such as concatenation, zero-or-more/one-or-more and negation of paths and more. For the purposes of finding interesting paths, property paths suffer from two issues, one of which cannot easily be resolved:

Wild-card Predicates - As of v1.1 property paths do not support wild-card predicates. This can be resolved by creating paths on the form of (predicate | !predicate)

Path returned - Property paths returns what is at the end of a given path, rather than the actual path itself. As the steps in-between the start and end node are crucial to the algorithm, this renders property paths unfit for use in [Interesting Path](#).

Property paths have therefore been discounted as a solution tool.

All [SPARQL](#) expressions are generated using *SPARQLFactory*, a custom class created to ease working with [StringTemplates](#) through a set of static methods. The [SPARQL](#) expressions for paths with k intermediate nodes, is generated by *GenerateQueries* which constructs [StringTemplates](#) for each of the four types of paths. The call hierarchy for *GenerateQueries* is illustrated in fig. 2.17 For brevity only

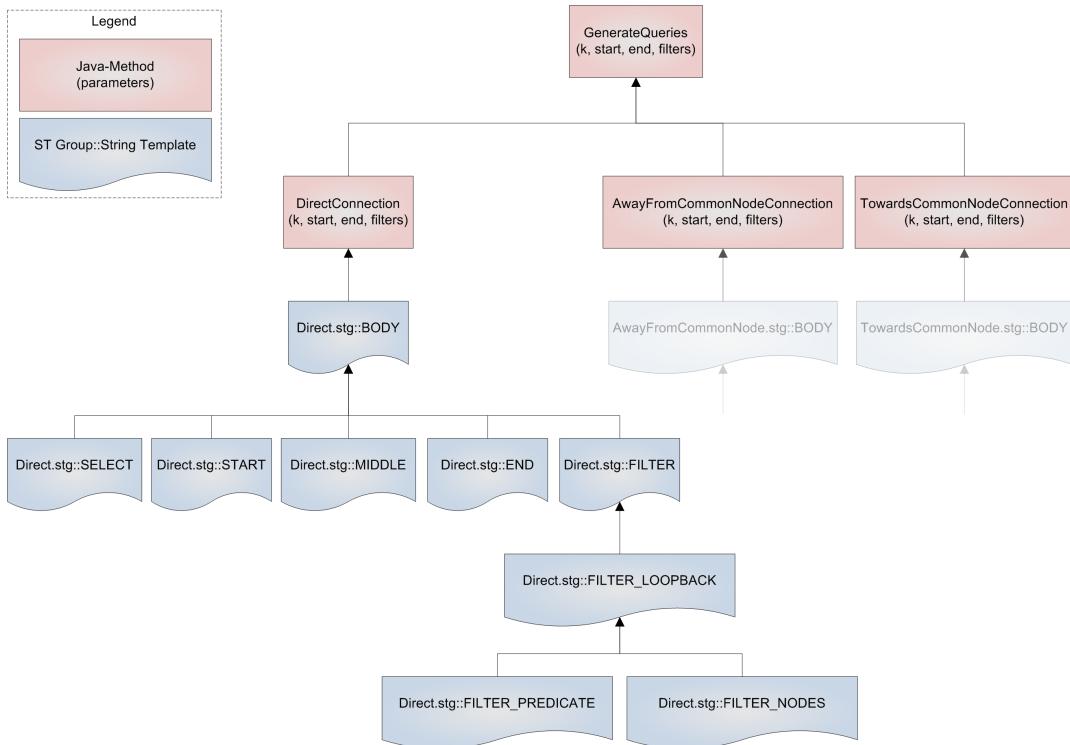


Figure 2.17: Call hierarchy for *GenerateQueries* showing the template calls

the direct connection call stack is shown in fig. 2.17. *GenerateQueries* returns an array of [SPARQL](#) expressions, created by three methods (*DirectConnection* creates [SPARQL](#) queries selecting paths from A to B and B to A), each of which calls the appropriate templates. Each template is broken down into sub-templates which handle selecting the appropriate variables, creating the right triple patterns, and apply

filtering to remove unwanted namespaces. Repeated vertices are prevented as shown in listing 2.25

Listing 2.25: FILTER_LOOPBACK prevents repeated vertices using the **SPARQL IN operator**

```

1 FILTER_LOOPBACK(start, end, iArray, j) ::==
2 <<
3 FILTER(?o<j> NOT IN (\<<start>\>, \<<end>\>><iArray: {k|,
    ?o<k>}>)) .
4 >>
```

Each triple pattern is explicitly prevented from containing vertices found in the other triples or the start and end vertices.

Once all queries have been created, they are executed on a data source. Initial attempts with running the queries directly on endpoints worked well for short paths(one and two intermediate nodes) but would result in time-outs for longer paths. To resolve this issue, **TDBs** are generated from endpoints and **SPARQL** expressions are executed against the local **TDB** model instead, which has the additional benefit of reuseability. For details on TDB see chapter [B](#)

Scoring and Natural Language

Once the possible paths have been extracted, a score has to be assigned to each path. During development it was noted that there is a large discrepancy between the set of intermediate nodes in all paths, and the set of distinct nodes in all paths. This discrepancy is often of an order of magnitude or more, which intuitively makes sense as each vertex can have several possible paths running through it. Literals are therefore only extracted for the discrete set of nodes, and stored in a one-to-many(one **URI** to zero or more literals) relation list using the [Google Guava v1.7](#) library. The literals are extracted in batches by the string template seen in listing 2.26

Listing 2.26: Literals are fetched in batches based on [URIs](#) and language

```

1 GET_LITERALs(sbj, lang) ::==
2 <<
3 SELECT DISTINCT ?s ?o WHERE
4 {
5     ?s ?p ?o .
6     FILTER(isLiteral(?o)) .
7     FILTER(?s IN (<sbj: {sbj| <sbj>}>)) .
8     <if(lang)>
9     FILTER(lang(?o) = '<lang>') .
10    <endif>
11 }
12 >>

```

Once literals have been collected, they are split into first sentences, then terms and finally stemmed using the [Stanford Core NLP](#)(for details about this process see chapter [C](#)).

Each path is scored based on a simple algorithm. First we define a few terms:

- $P_j \rightarrow$ The j-th path
- $I_{ji} \rightarrow$ Intermediate node i of path j
- $T \rightarrow$ bag-of-words set for search terms
- $L(I_{ji}) \rightarrow$ Function returning a bag-of-words set of literals for I_{ji}
- $H_{ji} \rightarrow$ The number of search term hits for intermediate node i in path j
- $IS_{ji} \rightarrow$ Term normalized score for intermediate node i in path j
- $PS_j \rightarrow$ Normalized score for path j

Each intermediate node is assigned a score based on eqn. [2.4](#) and [2.5](#)

$$H_{ji} \rightarrow T \cap L(I_{ji}) \quad H_{ji} \in \mathbb{N}, H_{ji} \in [0; |T|] \quad (2.4)$$

$$IS_{ji} \rightarrow \frac{H_{ji}}{|T|} \quad IS_{ji} \in \mathbb{R}, IS_{ji} \in [0; 1] \quad (2.5)$$

Eqn. [2.5](#) normalizes individual nodes in order to prevent the over-estimation of any one node in a path. And the final score for the individual path is calculated using eqn. [2.6](#)

$$PS_j = \frac{\sum_{i=1}^{|I_j|} IS_{ji}}{|I_j|} \quad (2.6)$$

Eqn. [2.6](#) defines the score of a path, PS_j as the sum of scores for all intermediate nodes in the path, normalized by the number of intermediate nodes. As a last step

the list of paths are sorted in descending score and the top k paths are returned as a list of *InterestingPaths*(for class diagram of InterestingPath see section E.3) to be visualised.

Chapter 3

Experiments & Trials

3.1 Introduction

This section details the experiments that have been performed during the project period, both to validate search algorithms as well as to estimate run-time improvements to **PBBBFS**. Each experiment is followed by an analysis and discussion with potential improvements discussed in chapter 4.

3.2 Measuring Interestingness

Interesting Graphs and **Interesting Paths**, (sections 2.5.2 and 2.5.3), are essentially search algorithms that explore a section of a graph based on user-determined parameters, in an attempt to find vertices of interest to the user. The performance of search algorithms are distinctly user-centric, in that the goodness of an algorithms' result is solely dependent on whether the user finds the results relevant.

Common methods for measuring performance of search algorithms involve the creation of a **gold standard**, a human-annotated corpus of documents with which the output of the algorithm can be compared. There are several methods for calculating performance for a search system, most of which build on **precision** and **recall**[45, p. 142-150]

$$Precision = \frac{\#(relevant\ items\ retrieved)}{\#(retrieved\ items)} \quad (3.1)$$

$$Recall = \frac{\#(relevant\ items\ retrieved)}{\#(relevant\ items)} \quad (3.2)$$

These metrics are defined for un-ordered sets, but most modern algorithms use ranking in order to ensure that the most relevant results are displayed first. For ordered ranking-based systems, **precision** can be redefined in terms of **top k results**

$$Precision_k = \frac{\#(relevant\ items\ retrieved)}{\#(top\ k\ retrieved\ items)} \quad (3.3)$$

[Recall](#) cannot be redefined in terms of k , as the total set of relevant elements is still important in order to determine the completeness of the search. The man-hours necessary to calculate [precision](#) and [recall](#) are not equal. In order to calculate [precision](#) we need only to determine the number of relevant elements returned, whereas [recall](#) requires a full mapping of all relevant elements in the element-space. This quickly becomes an time-intensive task, as it is not uncommon for vertices to have neighbours in the hundreds of thousands at even three or four steps from the initial vertex. As the creation of a full [gold standard](#) for a realistic dataset is outside the scope of this project, only [precision](#) will be measured during experimentation. Scoring of experiments will be done by members of the research group, with conflicts resolved by a third party.

3.3 Interesting Paths

The first part of the experiments conducted on the implementations described in section 2.5 is on [Interesting Paths](#).

The first set of experiments performed on interesting paths(section 2.5.3) consists of five searches performed, called questions. Each search consisted of a start- and end-vertex, a maximum number of intermediate vertices, a set of topics and search terms derived from said topics. The experimental parameters can be seen in table 3.2 with the top 50 interesting paths being extracted for each set of parameters, and language being restricted to English. The [precision](#) for the experiment can be seen in fig. 3.1 and the numeric values in table 3.1

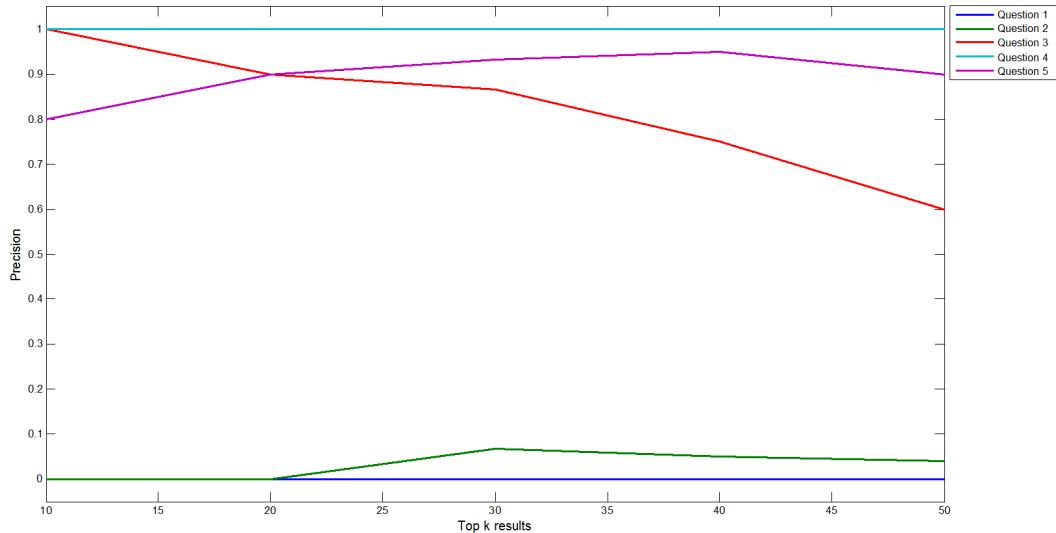


Figure 3.1: Precision results experiment

There are several interesting observations that can be made from this experiment.

Question	Paths Found	Precision at k				
		10	20	30	40	50
One	50	0	0	0	0	0
Two	24	0	0	0,067	0,05	0,04
Three	46	1	0,9	0,87	0,75	0,6
Four	50	1	1	1	1	1
Five	50	0,8	0,9	0,93	0,95	0,9

Table 3.1: Precision values from experiment

3.3.1 Question One

It would seem odd that there are no positive hits made between Tim Berners-Lee and Dijkstra. In order to determine if distance is the limiting factor, a search is performed from *Edsger W. Dijkstra to Resource Description Framework* with the same parameters given for question one. This search results in zero(0) paths, which validates the hypothesis.

3.3.2 Question Two

There were 24 paths found for this question, with 19 of 24 being given a positive score by the algorithm. This would indicate a productive search, but when investigating which search terms were most often found, terms such as *of* and *and* ranked very high. These terms are very frequent in English, and could therefore easily skew search-results by generating false-positive scoring. See section 3.3.6 for a discussion on methods mitigating this issue.

3.3.3 Question Three

For question three 17 out of 46 paths had a positive score, most of which walk through various programming languages, operating systems and computer scientists who have worked on programming languages. The latter paths mostly revolve around New York city and various military officers.

3.3.4 Question Four

Question four delivers an impressive average precision of one(1) for all five values of k. All returned paths have a positive score, with the lowest value being ~0.89. The algorithm has achieved this by returning a large group of *almost* similar paths, an example of which can be seen in listing 3.1

Listing 3.1: Sub-set of results from question four

```
PREFIX db:<http://dbpedia.org/>

<Path>
  <Step>db:resource/Bjarne_Stroustrup</Step>
  <Step>db:ontology/designer</Step>
  <Step>db:resource/C++</Step>
  <Step>db:ontology/influenced</Step>
  <Step>db:resource/Java_(programming_language)</Step>
  <Score>1.0</Score>
  <Done id="9"/>
</Path>
<Path>
  <Step>db:resource/Bjarne_Stroustrup</Step>
  <Step>db:ontology/designer</Step>
  <Step>db:resource/C++</Step>
  <Step>db:property/influenced</Step>
  <Step>db:resource/Java_(programming_language)</Step>
  <Score>1.0</Score>
  <Done id="10"/>
</Path>
<Path>
  <Step>db:resource/Bjarne_Stroustrup</Step>
  <Step>db:property/designer</Step>
  <Step>db:resource/C++</Step>
  <Step>db:ontology/influenced</Step>
  <Step>db:resource/Java_(programming_language)</Step>
  <Score>1.0</Score>
  <Done id="11"/>
</Path>
<Path>
  <Step>db:resource/Bjarne_Stroustrup</Step>
  <Step>db:property/designer</Step>
  <Step>db:resource/C++</Step>
  <Step>db:property/influenced</Step>
  <Step>db:resource/Java_(programming_language)</Step>
  <Score>1.0</Score>
  <Done id="12"/>
</Path>
```

Note that these four paths are essentially the same, the difference being whether the predicate is drawn from *property* or *ontology*, and thereafter every combination of these. So while in theory this provides a very high precision, a user would find the results less than optimal. We discuss this issue more in section 3.3.6

3.3.5 Question Five

All paths for question five have positive values and although there is some disagreement between the annotators and the algorithm, over all the results are good.

Start	End	Length	Topics	Search Terms
Tim Berners-Lee	Edsger W. Dijkstra	3	RDF, Semantic Web, Graph	Resource Description Framework, RDF, Graph, Semantic Web
Tim Berners-Lee	Grace Hopper	3	Compiler, web	COBOL, National Institute of Standards and Technology, compiler
Grace Hopper	Bjarne Stroustrup	3	Compiler, programming language	COBOL, C++, C
Bjarne Stroustrup	Java	3	Compiler, programming language	Java, C++, object oriented language
Alan Turing	Edsger W. Dijkstra	3	Turing Machine, graph theory	Automata, graph search, turing machine

Table 3.2: Parameters used during experimentation

Filter
http://www.w3.org/2002/07/owl#
http://www.w3.org/2000/01/rdf-schema#
http://www.w3.org/1999/02/22-rdf-syntax-ns#
http://dbpedia.org/class/yago/

Table 3.3: Filters applied during experiment

3.3.6 Reflections

The first experiment gave several insights on issues that need to be addressed in order to achieve better search-performance.

Stop Words: A common issue within the field of [information retrieval](#), usually defined as words that are so common as to add no semantic value to searches. Stop words can be dealt with using a *stop word list* or a *weighting scheme*[45, p. 25-26].

Stop word lists are generated in order to remove stop words from a dataset, thereby improving both memory and computation performance. This method has the distinct drawback of removing terms from searching, which can be problematic for certain phrases, such as *To be, or not to be*. Due to the general application area of GraphHelper, any stop word list would have to be dynamically generated, in order to be domain independent[63].

Weighting schemes assign a weight to terms based on some weighting system. A common model employed widely is the [vector space model](#)[45, p. 100-122], which constructs vectors from text in a [bag-of-words](#) manner which can then be compared in order to find similarities. Several of the more common weighting schemes up-prioritize rare terms and down-prioritize common terms, thereby lowering the effect of stop words when calculating similarities.

[Vector space model](#) and stop word lists can be used in conjunction, it would therefore be the project groups recommendation to experiment with both methods.

Path Duplicates: Showing the same path multiple times with only minor variation is unlikely to be well received by end users. The issue encountered in [DBpedia](#) is somewhat unusual, in that when a [Triplestore](#) contains ontologies with overlapping declarations, OWL:sameAs, OWL:equivalentProperty, OWL:equivalentClass is used to indicate communality in meaning. In the case of discovering interesting paths between *Bjarne Stroustrup* and the *Java Programming Language* as stated in section 3.3.4, the research group concludes that there must be a flaw in the dataset. A casual inspection of [DBpedia](#) suggests that it correctly implements OWL:sameAs for other overlaps between *db:property/* and *db:resource/*, and the research group has stumbled upon an exception.

In the general case, relying on [OWL Web Ontology Language](#) for mapping between ontologies would be a more robust methodology, but in special cases it might be necessary for users to create simple custom mappings in order to filter out oddities in a dataset, as the oddities described in section 3.3.4.

3.4 Interesting Graph

The first set of experiments performed on [Interesting Graphs](#)(section 2.5.2) consists of five searches, called questions. Each search consisted of a start vertex, a maximum number of steps out from the root vertex, a β value set by the user and a set of search terms. The experimental parameters can be seen in table 3.4 with at maximum

25 interesting nodes being selected for each set of parameters, and language being restricted to English. The [precision](#) for the experiment can be seen in fig. 3.2 and

Source	Length	β	Search Terms
Tim Berners-Lee	3	5	Resource Description Framework, RDF, Graph, Semantic Web
Grace Hopper	3	5	COBOL, Compiler, Business Oriented
Bjarne Stroustrup	3	5	C++, C, Embedded
Java	3	5	Object Oriented Language, Functional Language
Alan Turing	3	5	Automata, Graph Search, Turing Machine

Table 3.4: Parameters used during experimentation

the numeric values in table 3.5.

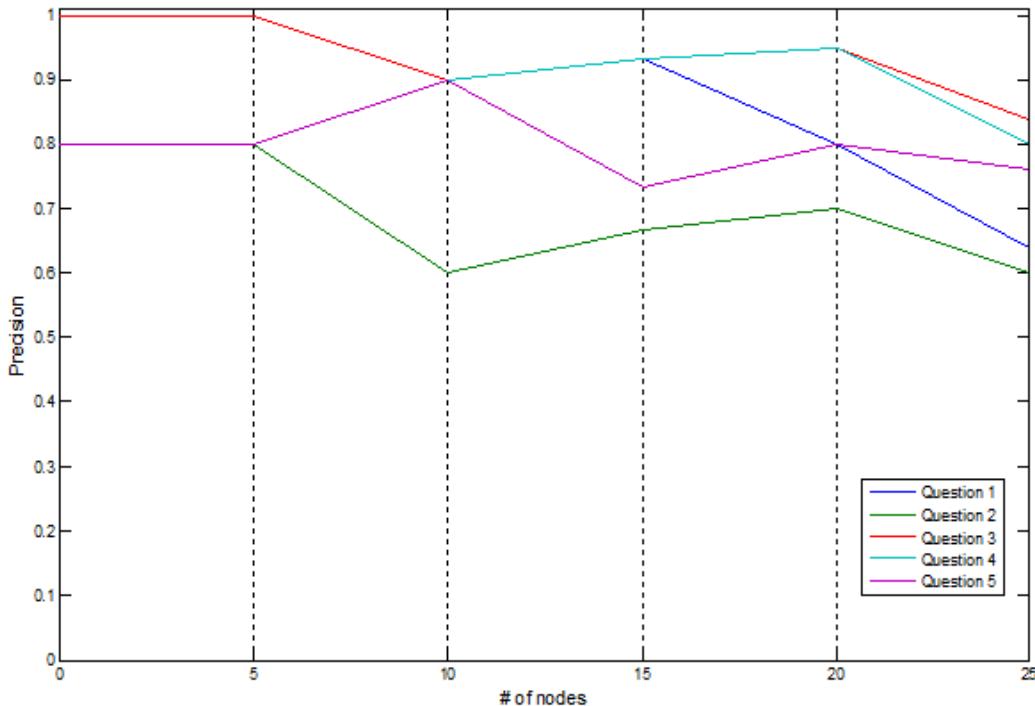


Figure 3.2: The precision of the interesting nodes in each subgraph

The source nodes in each sub-graph are not included when counting interesting nodes, as these nodes by default will be interesting.

3.4.1 Reflections

After performing this experiment several issues have been raised: All of the result-sets contain less than 25 interesting nodes. This seems unreasonable when considering

Question	Nodes Found	Precision at k						Mean
		5	10	15	20	25		
One	17	0.8	0.9	0.93	0.8	0.64	0.8122	
Two	22	0.8	0.6	0.6667	0.7	0.6	0.6944	
Three	22	1	0.9	0.93	0.95	0.84	0.9372	
Four	21	0.8	0.9	0.93	0.95	0.8	0.8639	
Five	24	0.8	0.9	0.73	0.8	0.76	0.7989	

Table 3.5: Precision values from experiment 1 for interesting graphs

that a full sub-graph, without the interesting node constraints, can contain hundreds of thousands of nodes when searching within 3 steps out from the initial node. Further investigation revealed that the low number of nodes returned, was the product of the termination conditions during the *RDFTree* building process, shown in listing 3.2

Listing 3.2: Code for building *RDFTree* before filtering - first version

```

1 % Terminates on first node dist _maxNodes
2 % _maxNodes is not a useful measure. Was dumped from
   later algorithms
3
4 while (myTree.getDepth() < _maxDist
5           && myTree.getCount() < _maxNodes
6           && myFrontier.size() > 0)
7 {
8     current = myFrontier.remove();
9     processedNodes.add(current.getMyValue());
10    processNeighbours(m, current, myFrontier, myTree,
11                      processedNodes, myScorer);
11 }
```

Listing 3.2 builds up the *RDFTree* as described in section 2.5.2.

The *_maxNodes* attribute was originally envisioned to allow a user to specify the maximum number of total nodes, interesting and uninteresting, returned by the algorithm. Implementing the restriction this early in the process, means that the termination conditions are very quickly met and once filtering has been applied to the *RDFTree*, only a sub-set of nodes will be left.

In addition the restriction on maximum depth should be changed so as to allow nodes up to the limit specified by the user, rather than less than. It is unlikely that this restriction had any effect on the experiment performed, given the much lower restriction imposed by *_maxNodes*.

The research group redesigned the process so that the graph tree had to be built containing all nodes not exceeding *maxDist*. Afterwards the nodes contained in the tree would be assigned a score of interestingness from the [Interesting Graph](#) algorithm, before filtering was applied. The code shown in listing 3.3 shows the new version for building the *RDFTree*. The process now continues to process nodes, as long as the frontier has candidates, but only candidates where the neighbours fit within the maximum distance restriction are actually processed.

Only after building the graph tree as seen in listing 3.3 and scoring all nodes contained in this graph tree, will the pruning of nodes take place and now the *maxNode* limit will be used, so the graph tree will only contain interesting nodes of an amount being equal or less than *maxNode*. The total number of nodes returned can be larger than *maxNode*, due to the necessity of keeping uninteresting nodes in order to connect to interesting nodes further down in the tree.

Listing 3.3: Code snippet for building up the graph tree before processing interesting nodes, v2

```

1 while (!myFrontier.isEmpty())
2 {
3     RDFTreeNode current = myFrontier.remove();
4     if((current.getDistance() + 1) <= myMaxDist)
5     {
6         processNeighbour(current);
7     }
8 }
```

The [precision](#) values for each question shown in table 3.5 is relatively high, with mean values ranging from 0.69 to 0.94. The [precision](#) values are highly dependent on the user choosing relevant search terms, and the data present in the [Triplestore](#). It is not easy to predict what effect the implementation errors discovered by the research group in [Interesting Graph](#) will have on future [precision](#) values. The nature by which [Interesting Graph](#) searches a graph, means that it might favour a node of particular interest at the end of an otherwise not particularly interesting path, or fully explore its closest neighbourhood before moving on. It is therefore recommended to repeat the experiment when the algorithmic defects have been fully addressed.

3.5 Parallel Bucket-Based Breadth First Search

In order to test the performance gains of the [PBBBFS](#) algorithm, a set of trial runs were run against [DBpedia](#). it is important to note that these trials are at best indicative, as there is a number of factors that are outside the search groups control.

DBpedia Load - There is no way to determine the load that DBpedia is under during experimentation.

Cache - Queries that have been cached will obviously be much faster if run a second time as the Triplestore need only read from memory. In order to mitigate this potential issue, one experiment is run at a in one day, thereby introducing a 24 hour delay between experiments.

Internet Delay/Speed - Queries were run from Aalborg University's network, which could be under various loads while running the experiment

With these caveats noted, the experiments were run once each for five URIs, with the parameters given in table 3.6 and filters in table 3.7

URI	# of Threads	Max Distance
Tim Berners-Lee	4	3
Grace Hopper	4	3
Bjarne Stroustrup	4	3
Java	4	3
Alan Turing	4	3

Table 3.6: Parameters for PBBBFS experiment

Filter
http://www.w3.org/2002/07/owl#
http://www.w3.org/2000/01/rdf-schema#
http://www.w3.org/1999/02/22-rdf-syntax-ns#
http://dbpedia.org/class/yago/

Table 3.7: Filters applied during experiment

PBBBFS is tested against the naïve BrFS algorithm initially developed, which fetches each URI individually. Sum time in milliseconds is measured for each of the five URIs, expanding the sub-graph from a distance from the initial node of zero to three. The timing values can be seen in table 3.8

URI	Naïve in ms(min)	PBBBFS in ms(min)	PBBBFS/Naïve
Tim Berners-Lee	3169239(~52)	785140(~14)	0.25
Grace Hopper	3010777(~50)	832724(~14)	0.28
Bjarne Stroustrup	3232623(~53)	793071(~12)	0.25
Java	3486162(~58)	872378(~14)	0.25
Alan Turing	2852315(~47)	713763(~11)	0.25

Table 3.8: Runtime values for the experiment

3.5.1 Reflections

PBBBFS presents across the board improvements when compared to naïve **BrFS**. Even with the caveats noted previously, the time reduction is quite impressive. Focusing in on *Bjarne Stroustrup*, results show that **PBBBFS** has grouped **URIs** into packages of 1 to 145 **URIs** per package. Plotting the unit size against the number of units with said size is shown in fig. 3.3

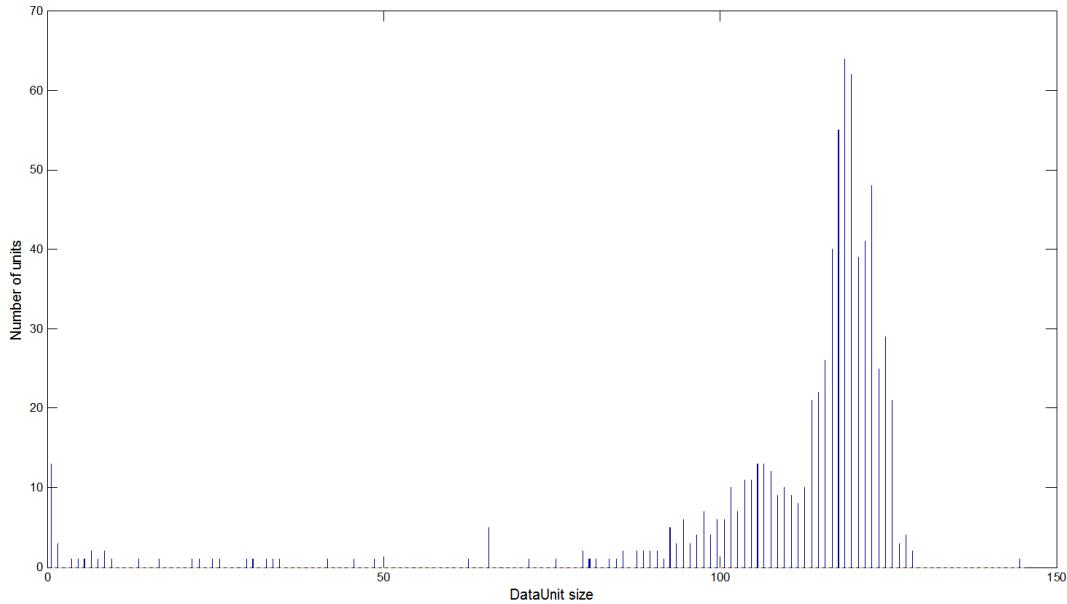


Figure 3.3: Number of packages grouped by number of **URIs** per package

Figure 3.3 shows that **PBBBFS** does a good job of grouping **URIs** into large units, with more than 87% of all packages containing 100 or more **URIs**. Further improvements should focus on optimally packing the units and hand-optimizing the code.

Chapter 4

Discussion

In this chapter limitations encountered throughout the project are discussed, as well as areas of future research and improvements. Improvements detailed apply to software design and implementation as well as algorithmic enhancements.

4.1 Software Architecture Status

As with most software projects involving frameworks, much has been learnt about [Jena](#) and [JGraphX](#) during the project period. It would therefore be advantageous to go back and apply some of the lessons learned to the early code developed, as well as refactoring and restructuring the code-base before implementing further features.

For long running processes, such as building [TDB](#) stores and scoring large graphs, the run-time can be fairly long. It would therefore be advantageous to show the status of these processes, such that a user can track the progress. This would require a more uniform threading approach, with well defined information flows to the [GUI](#)

4.1.1 Graph Visualisation

Most of the central features for visualising [RDF](#) graphs have already been implemented. Any new features are aimed at increasing usability, the major features being

Graph-Memory - Users can navigate between nodes by double clicking a desired node. Most users are used to having a *back button* with which to retrace their steps. Previous representations should be stored for a number of *steps* backwards.

Visual Appearance - [JGraphX](#) styles have been used extensively throughout the project, but in order to increase usability a centralized repository should be designed. All visualisations should draw styles from there, as well as update

their own appearance when changes are made to the repository. The repository should also be aware of namespaces, such that all elements belonging to <http://dbpedia.org/> are assigned a unique color, line type or shape. This can be implemented using standard [Swing](#) components, as well as creating events for notifying existing visualisations of changes.

4.1.2 Addressing Blank Nodes in Endpoints

As described in section 2.3.3, addressing blank nodes in remote [Triplestores](#) via [Jena](#) is problematic, as the [Triplestore](#) blank node ID is stripped during extraction and replaced with a local ID. If users are to be able to centre on blank nodes in endpoints, a method for addressing them will have to be developed. One option is to use indirect addressing, where all inbound and outbound nodes of the blank node is included in a [SPARQL](#) query. This method is not perfect, as there is no guarantee that two blank nodes don't share the same neighbours, it will therefore be necessary to do post-extraction processing to determine the uniqueness of the blank node. It would also require keeping an additional layer of neighbours in memory upon extraction, in order to have sufficient information to correctly identify blank nodes selected.

4.2 Interesting Path & Graph

While [Interesting Graph](#) and [Interesting Path](#) show generally promising [precision](#) values in sections 3.3 and 3.4, there is still much room for improvement. While the [precision](#) scores calculated give some indications of the performance of the algorithm, the lack of a more robust [gold standard](#) to do comparisons against makes it difficult to get a clear picture of how well the algorithms are performing. Even without this [gold standard](#) there are several areas where further research could enhance the algorithms.

4.2.1 Semantic Improvements

The current implementation only uses semantic information in order to calculate scores. This is not uncommon, as many information retrieval systems rely heavily on semantic information due to the relative simple nature of extracting meaning from semantic information, compared with syntactic information. In addition, only semantic information from literals is used, ignoring predicate labels and vertex [URIs](#). While some semantic data could be extracted from predicate labels, vertex [URIs](#) are much more problematic. When creating an [RDF](#) graph, [URIs](#) for vertices need not contain any relevant information about the node in question. It is common for [URIs](#) to simply be defined using a hashed value ID, which contains no semantic value about the node at hand. Predicates and node [URIs](#) are therefore unlikely to be sources of additional semantic information. It is therefore important to improve the quality of the semantic data already available in order to improve the accuracy of the algorithms. There are three areas where the research group believes improvements can

be attained

Term Weighting

The current model for matching search terms to literal text is very simple. The lemma for terms in both the search query and documents are found and the intersection between these two sets is selected. But not all term matches are equal. Terms that are rare within the space of documents, should carry more significance as they point to a smaller, more specific sub-set of documents. By the same reasoning, common terms should carry less weight as they are much more ubiquitous. Within modern information retrieval, this issue is addressed by the [vector space model](#) and appropriate term weighting schema. While there would certainly be a penalty to run-time performance, ([vector space model](#) is fairly simple to multithread), the run-time cost would be minor when compared with the run-time of [PBBFBS](#). [Vector space model](#) has strong empirical evidence[58] documenting its effectiveness, making it a strong candidate for improving accuracy of [Interesting Graph](#) and [Interesting Path](#).

Improving User Input

The performance of information retrieval systems are closely tied to the quality of the search parameters entered by the user. Improvements can therefore often be achieved from user-feedback on search terms entered, a common feature in most modern search engines. The research group considers the following three methods potential candidates for feedback-systems

Spell-checking: Incorrect spelling and typos can result in searches missing key terms. But these errors can also generate much more subtle errors. When parsing a sentence [Stanford Core NLP \(SCNLP\)](#) first tags all terms with a word class such as noun, noun-plural, verb and so forth. Next the tagged sentence is parsed by a lexicalized dependency parser which, using a set of grammatical rules and statistical analysis, determines the grammatical structure of the sentence. Based on the [Part-Of-Speech \(POS\)-tags](#) and grammar analysis, the lemmatizer determine lemmas for each term. If either the [POS](#)-tagger or dependency-parser performs misinterpretations due to spelling-mistakes, the proper analysis of the rest of the sentence can be affected. A simple dictionary-based approach should eliminate a large percentage of errors.

Search-Term Feedback: [RDF](#) graphs are not easily accessible and understandable to human users, a problem to which this project has been aimed at mitigating. In

order to assist users in selecting appropriate **URIs** to base searches on, a disambiguation feature proposed by Heim et al[33] was implemented. Something similar could assist users in selecting appropriate search terms, by providing feedback on whether a search term is relevant within a specific graph. A full search of all literals within a graph to determine the relevance of a given search term is unlikely to be practical, but as the user has already specified the neighbourhood in which the search will be performed, a simple heuristic can be applied. Checking literals in the immediate neighbourhood, at distances one or two, could provide an indication of the prevalence of the search term entered. The process could be executed in one of two ways:

Client-Load - The client fetches all literals at a distance k from a given node. Each literal is processed using **SCNLP** and the search term hit rate is returned to the user.

Endpoint-Load - Offloading the search to the endpoint might be more efficient, but creates new issues. Just searching directly for the entered term will ignore other morphological forms of the term. It might be possible to mitigate this issue by selecting appropriate morphological forms from WordNet[48]

Research and experimentation will be needed in order to determine the best approach.

Graph-Term Memory: Both **Interesting Graph** and **Interesting Path** search for and parse literals as part of their exploration process. By tracking the occurrence of lemmas processed, a local meta-**RDF** graph could be constructed, linking lemmas to **URIs**. This **RDF** graph could then be used for quick look-up of search terms feedback to the user. In addition the graph could be explored for densities of search terms, regions of the graph that uses a particular search term frequently, thereby generating candidate **URIs** for additional searches. Some research has been done into density calculations in **RDF**-graphs[56], and grammatical-based **RDF** graphs[42] but more work will have to be done to create a suitable model.

Graph Information

As previously mentioned semantic information is used exclusively in the algorithms developed during this project. Additional values should be derivable from **RDF**'s graph structure, using graph theory, although very little research has been published in this area. Graves et al. published the article *A method to rank nodes in an RDF graph*[20], which describes a method for ranking nodes in an **RDF** graph using *closeness centrality*. The graph is viewed as an undirected graph and *all-pair shortest paths* are calculated. Each path is scored based on the edges involved in the path, in this case as the inverse frequency of the predicate label within the graph.

Combining this graph theory approach with the already developed semantic methods could be a fruitful area of research. The importance of a node within an **RDF**

graph could be used to weight the semantic value, with different centrality measures being used depending on whether a user is searching for outlier or central nodes. Researchers will have to explore which measures of centrality are useful, and how to best integrate the results with the semantic information.

4.3 Parallel Bucket-Based Breadth First Search

PBBBFS showed significant run-time improvements when compared to a naïve **BrFS** algorithm in section 3.5. Even with these improvements, retrieval times of 11-14 minutes would be considered cumbersome and slow by most users. There is therefore a strong incentive to improving the run-time performance of **PBBBFS**. There are still several improvements that can be achieved, both in terms of run-time and memory performance.

4.3.1 DataUnit Packing

Packing the DataUnits more densely would reduce the number of **SPARQL** calls necessary, a tactic which so far has yielded good performance improvements. This can be viewed as an instance of the *bin packing problem*, where the objective is to pack a number of goods into the least number of bins, given certain restrictions. *Bin packing* is NP-complete[59, p. 595-597], but there are several heuristic based algorithm, which promise no worse than 22% more bins than an optimal solution for one dimensional problems. Even though this bin packing problem is two dimensional(**URI** length, max result set), more sophisticated heuristics should provide some improvement.

4.3.2 Memory Management

Several undocumented attempts have been made to extract sub-graphs of depth four and give. These attempts have only been partially successful, with some attempts crashing due to **Java** memory restrictions. While the memory limits for **Java** could be increased, a better solution would be to improve **PBBBFS**s memory usage. **PBBBFS** *DataManager* class currently implements three queues and one black list

MetaQueue - Keeps track of **URIs** for which to fetch result set sizes

DataQueue - Keeps track of **URIs** for which to fetch results

DataQueue+1 - Queue for next iteration step outwards

Blacklist - A HashSet copy of **DataQueue** which is used for quick lookup of **URIs** for filtering

MetaQueue and *DataQueue* can easily be merged into a single structure, with no cost in performance. *Blacklist* can be merged with *MetaQueue* and *DataQueue*, but there will be a loss in performance depending on the data structure chosen. If the new structure is a HashSet, lookup times will be constant but iteration will suffer,

where as a list structure will have good iteration time but comparatively poor lookup time. The optimal solution will depend on the number of Contains/Iterations and their associated cost, which can be determined with a set of simple experiments.

4.3.3 Write-to-Disk

The current implementation of **PBBBFS** writes gathered triples to a disk-based **TDB** for persistent storage. This write is performed once for each *DataUnit*, which as it involves I/O activity, is a comparatively expensive operation. There are three alternative strategies available

Full-Model Collect triples until all frontiers have been explored to a depth of k .
Expensive memory usage strategy

Frontier-Move Collect triples until one frontier has been fully explored, where upon all triples are written to disk in bulk. Better memory usage strategy, but does not scale well for larger frontiers

Block-Based Collect triples until a certain limit is met, upon which the collection is exported to disk and the process starts anew. Potentially lowest memory footprint, but with increased cost to I/O. Best scaling of the three options

Option one and two suffer from scalability issues, in that triples for frontiers at depths of 4 or 5 can take up vast amounts of memory. Option three offers a good balance between memory usage and I/O cost.

With these improvements implemented, **PBBBFS** should improve significantly faster and scale better for bigger and deeper graphs.

Chapter 5

Conclusion

The problem statement set forth by the research group was the following:

How to develop a graph visualisation application with tools which assist end-users with the discovery of relevant information from Semantic Web sources

On the basis of this statement, the application *GraphHelper* was created. In order to speed up development, [Jena](#) and [JGraphX](#) frameworks were employed for graph visualisation and [RDF](#) data management. The current version of *GraphHelper* allows users to:

- Load and save [RDF](#)-files for all major file-formats, as well as to connect to remote [Triplestores](#) through web-based interfaces
- Probe local and remote [RDF](#)-sources through user-entered [SPARQL](#) expressions
- Visualise sub-graphs of larger complex [RDF](#)-graphs in a *spot-light* like manner, where a local neighbourhood of a node is shown. Users can navigate around the sub-graph by double-clicking on nodes in order to move the *spot-light* focus

The only feature specified in the problem statement that has not been implemented, is the ability to export graphs to images. In addition to these features two intelligent search algorithms have been developed, named [Interesting Graph](#) and [Interesting Path](#), as well as an algorithm for efficiently extracting sub-graphs, called [Parallel Bucket-Based Breadth First Search \(PBBBFS\)](#).

Interesting Graph is a novel A*-inspired BrFS algorithm for RDF-graphs. A user selects a node to explore, as well as a set of terms that describes a topic of interest, as well as a set of secondary parameters. The algorithm searches the neighbourhood of the selected node, looking for interesting nodes, and constructs a sub-graph to be displayed upon completion.

Interesting Graph has been tested against a real-life data set, called **DBpedia**. Five nodes were explored, each with associated search terms. The research group hand-annotated the result set in order to calculate the **precision** values. The best experiment obtained a **precision** of ~0.95, while the worst experiment obtained a **precision** of ~0.60 in spite of certain software defects. The research group considers these results promising and expect to be able improve the **precision** through error corrections in software and more advanced information retrieval models as described in section 4.2.

Interesting Path is an extension of previous research by Heim et al.[33]. A user selects a start node and an end node, as well as a set of terms that describes the topic of interest, as well as set of secondary parameters. The algorithm searches for paths connecting the start and end node with a predefined amount of intermediate nodes between them, and visualise the results in a graph-like manner, showing the most promising paths discovered between the start and end node containing interesting intermediate nodes.

Interesting Path has in the same manner as **Interesting Graph** been tested against the real-life data set, **DBpedia**. Five sets, each containing a start node and an end node were explored in order to find interesting paths between them. The research group hand annotated the result set in order to calculate the **precision** values. The best experiment obtained a **precision** of ~0.95, while the worst experiment obtained a **precision** value of ~0.00. The best-case results were over-estimated due to duplicate paths resulting from incorrectly-applied ontologies in **DBpedia**. The worst-case results were determined to stem from poor choice of search terms. For both of these issues the research group has proposed several methods to remedy these hindrances, and predicts that better results can be achieved.

PBBBFS is a novel BrFS algorithm devised by the research group in order to efficiently extract sub-graphs from remote endpoints. The development of this algorithm was necessary in order to overcome issues related to time-outs and run-time performance when working with large datasets. **PBBBFS** iteratively probes around a given node and groups neighbours into more optimal buckets of **URIs**, which can then be extracted more efficiently. In addition to the improvements in collection-speed, **PBBBFS** stores the resulting sub-graphs in local **Triplestores** in order to allow for fast searches of previously collected sub-graphs.

PBBBFS has been tested on the real-life dataset **DBpedia**. The algorithm was compared with a naïve BrFS implementation. **PBBBFS** extracted sub-graphs of equal size at ~4x the speed of the naïve algorithm.

The research group is satisfied with the improvements gained, but has outlined several suggestions for improving run-time and memory management for [PBBBFS](#).

In 2006 Nigel Shadbolt, Tim Berners-Lee and Wendy Hall published the article *The Semantic Web Revisited* in *IEEE Intelligent Systems*. In the article Shadbolt et al. discuss the slow uptake of [RDF](#), when compared to the explosive growth of the [WWW](#). The quotation below summarizes some of the issues noted for the slow rate of adoption of [RDF](#)[57]:

This next wave of data ubiquity will present us with substantial research challenges. How do we effectively query huge numbers of decentralized information repositories of varying scales? How do we align and map between ontologies? How do we construct a [Semantic Web](#) browser that effectively visualise and navigates the huge connected RDF graph? How do we establish trust and provenance of the content?

Great strides have been made within the areas of efficient querying of large decentralized datasets, with the modern [Triplestore](#) being able to manage up to a trillion triples[62]. Research into ontologies is also an active area of exploration, with thousands of articles being published on [RDF](#).

Visualisation and efficient presentation of [RDF](#) information to users however, appears to only enjoy sparse attention from the research community. The research group believes that in order for the [Semantic Web](#) to become accessible to non-experts, intelligent user-centric information retrieval systems need to be developed. The research group is confident that future research into this topic will bear fruit.

Appendix A

StringTemplate

[StringTemplate](#)[54] is a [Berkeley Software Distribution](#) licensed open source Java library, with ports for several other languages, aimed towards easing the creation of highly formatted text. Originally developed by Terence Parr[53] as part of the ANTLR project[52] to ease the process of generating formatted code created by ANTLR. [StringTemplate](#) has since been used in various other scenarios, such as dynamic web-site generation etc. This section will provide a brief introduction into using [StringTemplate](#). For a more comprehensive overview of [StringTemplate](#) abilities, please see the StringTemplate 4 documentation[30]

[StringTemplate](#) consists of three components:

ST Group An ST Group is a file with the extension .stg which contains one or more templates. These templates can be used in conjunction to create complex results

Template A template function with simple embedded logic

Java Code To create, fill in and generate a string from a template

Templates are written in plain text, which defines both a format, as well as the variables which will be inserted into it. Listing A.1 shows a simple example

Listing A.1: A simple template

```
1 INBOUND (URI) ::=  
2 <<  
3 SELECT ?s ?p (\<<URI>\> AS ?known) WHERE  
4 {  
5   ?s ?p \<<URI>\> .  
6 }  
7 LIMIT 10000  
8 >>
```

Line 1 contains the template name, *INBOUND*, as well as the input **URI**. **URI** could be a primitive, list, array, custom object or even another template. In this case it is the place-holder for an **URI** string, which when inserted will generate a simple **SPARQL** SELECT statement. Inputs within a template are encased using triangles(< and >), and since **URIs** within **SPARQL** are also encased in triangles, an extra set is escaped around **URI**. **StringTemplate** does, by default, maintain a template's format which is advantageous for generating human-readable strings.

The **Java** code to invoke listing A.1 can be seen in listing A.2

Listing A.2: Java code to invoke template

```

1 String URI = ...;
2
3 STGroup myGroup = new STGroupFile("STFile.stg");
4 ST inbound = myGroup.getInstanceOf("INBOUND");
5 inbound.add("URI", URI);
6
7 String result = inbound.render();

```

Line 3 loads the ST group from a file *STFile.stg* and line 4 creates an instance of the template *INBOUND*. Line 5 adds the string *URI* to the template, and finally the template is generated in line 7. **StringTemplate** uses a form of **lazy evaluation**, and it is therefore only when *render()* is called that the actual string is created.

As previously mentioned **StringTemplates** can handle lists and arrays. Listing A.3 shows a template which generates a dynamic list of **SPARQL** filters.

Listing A.3: A simple template

```

1 FILTER(filterURIs) ::==
2 <<
3 <filterURIs:
4 {f|
5 FILTER(!strStarts(STR(?p), "<f>")) .
6 FILTER(!strStarts(STR(?s), "<f>")) .
7 }>
8 >>

```

Line 3-7 is a form of *for each* loop, with the temporary variable *f* being inserted in line 5 and 6 on each iteration. The resulting **SPARQL** expression would filter out *?p* and *?s* for **URIs** starting with any string found in *filterURIs*.

Nesting templates within templates is one of the strong features of **StringTemplate**. By partitioning a text into smaller templates, it is easier for a designer to create complex strings. By extending upon listing A.1 we can combine it with listing A.3 to create a simple **SPARQL** generator with filtering, as seen in listing A.4.

Listing A.4: Templates can be nested within templates

```

1 INBOUND (URI, filters, limit) ::= 
2 <<
3 SELECT ?s ?p (\<<URI>\> AS ?known) WHERE
4 {
5   ?s ?p \<<URI>\> .
6   <filter>
7 }
8 <if(limit)>LIMIT <limit><endif>
9 >>
```

Listing A.4 can accept filters and if no filters are added, line 6 will simply not be shown. Line 8 inserts a limit, but if no limit is inserted it is necessary to perform an if-check first, so as to not print *LIMIT blank*. If-checks in [StringTemplates](#) can be considered *is-set* checks in that they return true if a variable is present, and false if null.

The [Java](#) code to generate listing A.4 can be seen in listing A.5

Listing A.5: Java code to invoke templates

```

1 String URI = ...;
2 String[] filterURIs = ...;
3 int limit = ...;
4
5 STGroup myGroup = new STGroupFile("STFile.stg");
6
7 ST STfilter = myGroup.getInstanceOf("FILTER");
8 STfilter.add("filterURIs", filterURIs);
9
10 ST STinbound = myGroup.getInstanceOf("INBOUND");
11 STinbound.add("URI", URI);
12 STinbound.add("filters", STfilter);
13 STinbound.add("limit", limit);
14
15 String result = STinbound.render();
```

[StringTemplate](#) has been used extensively within this project to generate custom SPARQL expressions.

Appendix B

TDB

TDB is a file-based **RDF** store developed as part of the Apache **Jena** project[40]. **TDB** is related to **Fuseki**[39], but is only intended for single client usage, and does therefore not have **REST**-style **SPARQL** support. A **TDB** storage unit can be interacted with by using either command-line scripts or the **Jena API**, but instructions will be given for the **Java API** only in this report.

TDB triple stores are created using by using the **TDBFactory** as seen in listing B.1

Listing B.1: **TDBFactory** creates the required files

```
1 String dir = "TDBStore";
2 Dataset dataset = TDBFactory.createDataset(dir);
```

If the *dir* location already contains a valid **TDB** triple store, this store will be returned by the method call. Each **TDB** contains three hash-files, Subject-Predicate-Object(SPO), Predicate-Object-Subject(POS) and Object-Subject-Predicate(OSP), as well as transaction logs, prefix mappings and other meta data information. Once a store has been created, it can contain multiple *models*, all accessible through the *Dataset*. The recommended method for interacting with models, is via transactions. Listing B.2 shows a code snippet where a set of triples are added to the default graph of a **TDB** triple store.

Listing B.2: Bulk insert of triples using transactions

```

1 String dir = "TDBStore";
2 Dataset dataset = TDBFactory.createDataset(dir);
3 List<Statement> stmnts = ...
4
5 dataset.begin(ReadWrite.READ);
6 Model model = dataset.getDefaultModel();
7 model.add(stmnts);
8 dataset.commit();
9 dataset.end();

```

Once a model has been loaded, it can be interacted with in much the same way as a [Jena](#) standard models. Listing B.3 shows a [SPARQL](#) expression being executed upon a [TDB](#) model.

Listing B.3: SPARQL expression executed on TDB Model

```

1 String query = ...
2 String dir = "TDBStore";
3 Dataset dataset = TDBFactory.createDataset(dir);
4 Model model = dataset.getDefaultModel();
5
6 dataset.begin(ReadWrite.READ);
7 Query q = QueryFactory.create(query);
8 QueryExecution qExe = QueryExecutionFactory.create(q,
    model);
9 rs = qExe.execSelect();
10 // process results
11 dataset.end();

```

B.1 Concurrency

[TDB](#) triple stores, and by extension [Jena](#) models, are not inherently thread-safe, and concurrency therefore has to be handled explicitly. The [Jena API](#) concurrency model implements a [Multiple-Reader/Single-Writer \(MRSW\)](#)[38] model, using indicative locking. Listing B.4 shows a model being locked for read-access.

Listing B.4: A read lock applied to a TDB model

```

1 String query = ...
2 String dir = "TDBStore";
3 Dataset dataset = TDBFactory.createDataset(dir);
4 Model model = dataset.getDefaultModel();
5
6 model.enterCriticalSection(Lock.READ);
7 // perform read-action
8 model.leaveCriticalSection();

```

The lock system is a contract, and does therefore not enforce read/write within the critical sections. It only ensures that no critical section declared as write is called when other threads are reading, and performing writes in a read section should be avoided. It is worth noting that iterators cannot safely be read when outside the critical area, as an iterator is not guaranteed to contain consistent data once leaving the critical area.

B.2 TDB Management

The persistent nature of TDB triplestores allows for reuse of already collected sub-graphs. A sub-graph can be reused by a new query if

$$\begin{aligned}
 \text{Source}_{TDB} &= \text{Source}_{new} \\
 \text{Endpoint}_{TDB} &= \text{Endpoint}_{new} \\
 \text{Filters}_{TDB} &\supseteq \text{Filters}_{new}
 \end{aligned} \tag{B.1}$$

Sub-graphs generated for [Interesting Graphs](#) and [Interesting Paths](#) can be reused interchangeably, where either *source* or *target* for an [Interesting Path](#) can be reused by an [Interesting Graph](#) and vice versa. Interaction with [TDBs](#) is handled by the custom class *TDBManager*(for class diagram see fig. [E.1](#)). All [TDBs](#) are stored in the sub-folders under *TDBStorage* and in addition to the [TDB](#) files, each [TDB](#) folder has an XML settings file as well as a text file containing the sub-graphs frontier. *TDBManager* probes these [XML](#) files in order to find suitable [TDBs](#) based on the requirements set forth in [B.1](#). If a [TDB](#) is found that meets the requirements but is not of sufficient size, *TDBManager* will regenerate the frontier and expand the [TDB](#) with the necessary number of steps.

Stanford Core NLP

Searching through and working with natural language can be very complex, due to the varied ways in which human write. When searching for the term, *organize* a user would expect the system to return documents containing morphologicals of the search term, such as *organizes* and *organizing*. There are two methods for reducing terms to a more common form, namely *stemming* or *lemmatization*[45, p. 30-33].

Stemming is a set of several different methods that employ heuristics in order to reduce words. Stemming approaches vary greatly by language, but most rely on defining a set of simple rules which can be executed iteratively.

Lemmatization attempts to reduce words using morphological analysis in order to reduce words to their dictionary definition, called *lemma*. The [SCNLP](#) library implements a lemmatization approach, which involves four step.

Tokenize - The process of splitting a document up into tokens[27].

Sentence Split - Breaks a document down into sentences.

Part-of-Speech(POS) - Annotates all tokens with with their grammatical word-class[28]

Lemmatization - Finds the lemma for tokens using the semantic and syntactic information derived in the previous steps

All interaction with [SCNLP](#) is handled by the class *WordScorer*, (Class diagram section [E.4](#)). *WordScorer* configures Stanford Core's pipe system with the selected options before processing text strings. Listing [C.1](#) shows *WordScorer* parsing search terms into lemmas

Listing C.1: WordScorer breaks down search terms into lemmas

```

1 myBagOfWordsProperties.put("annotators", "tokenize,
    ssplit, pos, lemma");
2 myPipes = new StanfordCoreNLP(myBagOfWordsProperties);
3 Annotation doc = new Annotation(mySearchString);
4 myPipes.annotate(doc);
5
6 for (CoreMap sentence :
    doc.get(SentencesAnnotation.class)) {
7     for (CoreLabel token :
        sentence.get(TokensAnnotation.class)) {
8         String lemma = token.get(LemmaAnnotation.class);
9         if (!myLemmas.contains(lemma.toLowerCase())) {
10             myLemmas.add(lemma.toLowerCase());
11         }
12     }
13 }
```

Lines 1-4 configures Stanford Core NLP to perform the four necessary steps as well as creating an annotated document of the search string. Lines 6-13 iterate over the lemmas and selects all distinct lemmas.

Once the search terms have been lemmatized, *WordScorer* is ready to score documents. The process for scoring documents is much the same as for preparing search terms. Listing C.2 illustrates this

Listing C.2: WordScorer scores documents and collections of documents

```

1 public double ScoreDocuments(Collection<String> _docs) {
2     int score = 0;
3     int lemmaSpace = myLemmas.size();
4     List<String> tempLemmas = new LinkedList<>(myLemmas);
5
6     for (String d : _docs) {
7         score += ScoreDocument(d, tempLemmas);
8     }
9     return betaValue
10        * (Math.log((double) (1 + score))
11        / Math.log((double) (1 + lemmaSpace)));
12 }
```

Listing C.2 iterates across the collection, scoring each document individually. Search term lemmas that are matched are removed from the collection, so the maximum score possible is

$$Score_{max} = \beta \cdot \frac{\log(1 + score)}{\log(1 + |lemmas|)} \Big|_{score=|lemmas|} = \beta \cdot 1 \quad (\text{C.1})$$

Some *WordScorer* methods do not use beta-values, and the result range is therefore [0; 1].

Appendix D

User Assisted Search

In several cases the user has to specify a vertex from which to search. This assumes that the user already knows the correct [URI](#) which identifies the desired topic, but the identification of said [URI](#) can be problematic for large and complex [Triplestores](#). In order to assist the user in this initial step, several dialog windows in GraphHelper offers a *disambiguity* feature. Figure D.1 shows an example of such a disambiguity search. The user can enter terms to search for as well as an optional language

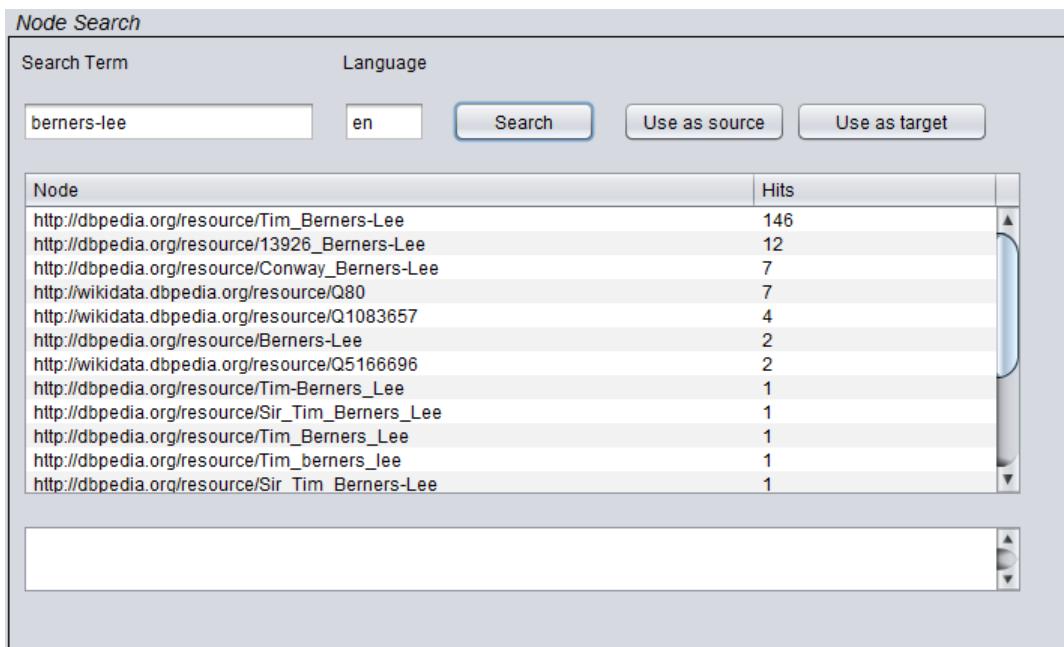


Figure D.1: Disambiguation segment from Interesting Path Dialog

restriction. In this case we are searching for *berners-lee* and restricting language to *English*. The search button executes a [SwingWorker](#) thread which runs the [SPARQL](#)

expression seen in chapter D[33, p.3]

```

1 SELECT ?s ?l count(?s) as ?count WHERE {
2   ?someobj ?p ?s .
3   ?s <http://www.w3.org/2000/01/rdf-schema#label> ?l .
4   ?l bif:contains '"berners-lee"' .
5   FILTER (!regex(str(?s),
6     '^http://dbpedia.org/resource/Category:' )) .
7   FILTER (!regex(str(?s),
8     '^http://dbpedia.org/resource/List' )) .
9   FILTER (!regex(str(?s), '^http://sw.opencyc.org/' )) .
10  FILTER (lang(?l) = 'en') .
11  FILTER (!isLiteral(?someobj)) .
12 } ORDER BY DESC(?count) LIMIT 20

```

The expression selects all vertices with RDF-schema#label literals containing the search term. The number of hits is counted by vertex, which can be considered a simple measure of importance. Finally the results are ordered in descending order and the top twenty vertices and accompanying scores are returned to be displayed for the user.

It should be noted that the current implementation of the disambiguity search is aimed towards [DBpedia](#) and [Open Link Virtuoso](#). In line 4 the [Open Link Virtuoso](#) custom command *bif:contains* is used due to the improved performance, but it can be replaced with the generic [SPARQL](#) query: **FILTER(CONTAINS(?l, "berners-lee"))** .

Appendix E

Class Diagrams

This chapter contains full class diagrams for key parts of the GraphHelper implementation

E.1 Parallel Bucket-Based Breadth First Search



Figure E.1: Full class diagram for the PBBBFS algorithm

E.2 Visualisation Classes



Figure E.2: Explicit class diagram for visualisation classes

E.3 Data Containers



Figure E.3: Data containers used to decouple data management from visualisation

E.4 RDFTree & WordScorer

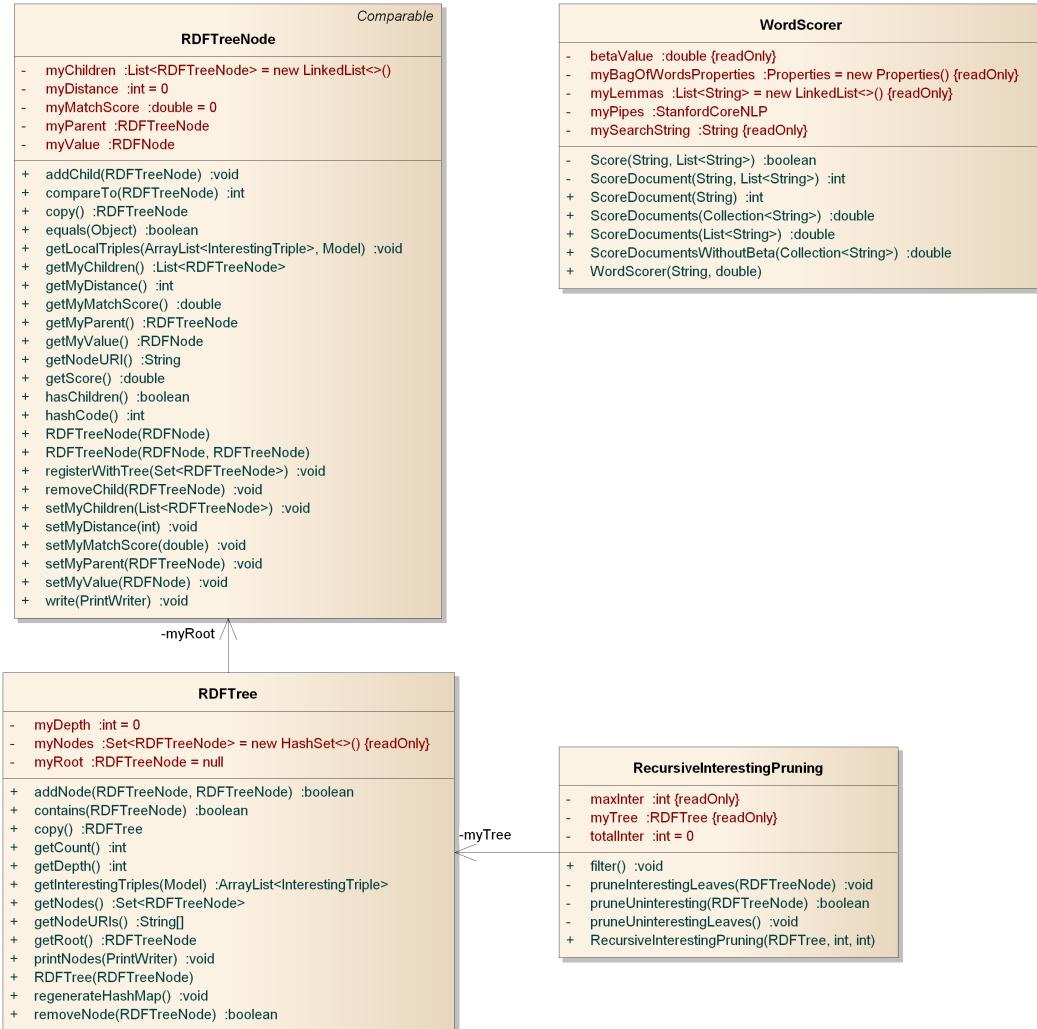


Figure E.4: RDFTree, node, filter and WordScorer classes

E.5 Data Sources



Figure E.5: DataSourceInterface, GenericHTTPEndPoint and LocalGraph classes

E.6 Data Storage



Figure E.6: DataStorage and DataRepositoryInterface classes

Bibliography

- [1] [http://dbpedia.org/.](http://dbpedia.org/)
- [2] David Beckett and Tim Berners-Lee. Turtle - terse rdf triple language. <http://www.w3.org/TeamSubmission/turtle/#sec-intro>.
- [3] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.
- [4] World Wide Web Consortium. N-triples. <http://www.w3.org/2001/sw/RDFCore/ntriples/>.
- [5] World Wide Web Consortium. Notation 3. <http://www.w3.org/TeamSubmission/n3/>.
- [6] World Wide Web Consortium. Owl 2 web ontology language. <http://www.w3.org/TR/owl2-overview/#Introduction>.
- [7] World Wide Web Consortium. Owl web ontology language. <http://www.w3.org/TR/2004/REC-owl-guide-20040210/#Introduction>.
- [8] World Wide Web Consortium. Rdf/xml syntax specification (revised). <http://www.w3.org/TR/REC-rdf-syntax/>.
- [9] World Wide Web Consortium. Resource description framework(rdf) model and syntax specification. <http://www.w3.org/TR/1999/PR-rdf-syntax-19990105/>.
- [10] Ron Davidson and David Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, October 1996.
- [11] Bob DuCharme. *Learning SPARQL*. O'Reilly Media, Inc, Sebastopol, 2011.
- [12] GraphX for .NET. Graphx for .net group. <http://www.panthernet.ru/en/projects-en/graphx-en>.

- [13] Python Software Foundation. Python programming language - official website. <http://www.python.org/>.
- [14] Jena .NET Framework. Jena .net framework. <http://www.linkeddatatools.com/downloads/jena-net>.
- [15] RDFLib FuXi. Rdflib fuxi. <http://code.google.com/p/fuxi/>.
- [16] Google. Guava: Google core libraries for java 1.6+. <https://code.google.com/p/guava-libraries/>.
- [17] Sunil Goyal and Rupert Westenthaler. Rdf-gravity. <http://semweb.salzburgresearch.at/apps/rdf-gravity/>.
- [18] Laura A. Granka, Thorsten Joachims, and Geri Gay. Eye-tracking analysis of user behavior in www search. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '04*, pages 478–479, New York, NY, USA, 2004. ACM.
- [19] Alvaro Graves. Visual representation of rdf. <https://github.com/alanagrafu/visualRDF>.
- [20] Alvaro Graves, Sibel Adali, and Jim Hendler. A method to rank nodes in an rdf graph. In Christian Bizer and Anupam Joshi, editors, *International Semantic Web Conference (Posters & Demos)*, volume 401 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
- [21] Jgraphx Group. Jgraphx group. <https://github.com/jgraph/jgraphx>.
- [22] JUNG Group. Jung - java universal network/graph framework. <http://jung.sourceforge.net/>.
- [23] NetworkX group. Networkx for python. <http://networkx.github.io/>.
- [24] Prefuse Group. prefuse | interactive information visualization toolkit. <http://prefuse.org/>.
- [25] Project Group. Retrieving blank node mapping. [http://stackoverflow.com/questions/22536775/retrieving-blank-node-mapping/](http://stackoverflow.com/questions/22536775/retrieving-blank-node-mapping).
- [26] RDFLib group. Rdflib. <https://github.com/RDFLib>.
- [27] Stanford NLP Group. The stanford nlp (natural language processing) group. <http://nlp.stanford.edu/software/tokenizer.shtml>.
- [28] Stanford NLP Group. The stanford nlp (natural language processing) group. <http://nlp.stanford.edu/software/tagger.shtml>.
- [29] Stanford NLP Group. The stanford nlp (natural language processing) group. <http://nlp.stanford.edu/software/corenlp.shtml>.

- [30] StringTemplate Developer Group. Stringtemplate 4 documentation. <https://theantlrguy.atlassian.net/wiki/display/ST4/StringTemplate+4+Documentation>.
- [31] The W3C SPARQL Working Group. Sparql 1.1 overview. <http://www.w3.org/TR/sparql11-overview/>.
- [32] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.
- [33] Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. Relfinder: Revealing relationships in rdf knowledge bases. In *Proceedings of the 4th International Conference on Semantic and Digital Media Technologies: Semantic Multimedia*, SAMT '09, pages 182–187, Berlin, Heidelberg, 2009. Springer-Verlag.
- [34] Philipp Heim, Jürgen Ziegler, and Steffen Lohmann. gFacet: A browser for the web of data. In *Proceedings of the International Workshop on Interacting with Multimedia Content in the Social Semantic Web (IMC-SSW 2008)*, volume 417 of *CEUR-WS*, pages 49–58, 2008.
- [35] igraph group. The igraph library. <http://igraph.sourceforge.net/index.html>.
- [36] Franz Inc. Gruff: A grapher-based triple-store browser for allegrograph. <http://franz.com/agraph/gruff/>.
- [37] Apache Jena. Apache jena. <http://jena.apache.org/>.
- [38] Apache Jena. Apache jena - concurrency. <https://jena.apache.org/documentation/notes/concurrency-howto.html>.
- [39] Apache Jena. Apache jena - fuseki. http://jena.apache.org/documentation/serving_data/.
- [40] Apache Jena. Apache jena - tdb. <http://jena.apache.org/documentation/tdb/index.html>.
- [41] jena.apache.com. getting started, October 2013.
- [42] Claudia Kunze and Lothar Lemnitzer. Germanet - representation, visualization, application. In *LREC*. European Language Resources Association, 2002.
- [43] Charles E. Leiserson. Concepts in multicore programming: Lab 4: Breadth-first search. <http://courses.csail.mit.edu/6.884/spring10/labs/lab4.pdf>.

- [44] JGraph Ltd. Jgraphx user manual. http://jgraph.github.io/mxgraph/docs/manual_javavis.html.
- [45] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
- [46] Stefano Mazzocchi and Paolo Ciccarese. Simile | welkin. <http://simile.mit.edu/welkin/>.
- [47] Microsoft. Automatic graph layout. <http://research.microsoft.com/en-us/downloads/f1303e46-965f-401a-87c3-34e1331d32c5/default.aspx>.
- [48] George A. Miller. Wordnet: A lexical database for english. *COMMUNICATIONS OF THE ACM*, 38:39–41, 1995.
- [49] Enrico Minack. dotsesame. <http://sourceforge.net/projects/dotsesame/>.
- [50] Oracle. Java.com. <http://www.java.com/en/>.
- [51] palesz. Graph#. <http://graphsharp.codeplex.com/>.
- [52] Terence Parr. Antlr. <http://www.antlr.org/>.
- [53] Terence Parr. Professor terence parr – university of san francisco. <http://parrt.cs.usfca.edu/>.
- [54] Terence Parr. Stringtemplate. <http://www.stringtemplate.org/>.
- [55] The Mono Project. Mono. http://www.mono-project.com/Main_Page.
- [56] Dongmei Ren, Baoying Wang, and William Perrizo. Rdf: A density-based outlier detection method using vertical data representation. In *ICDM*, pages 503–506, 2004.
- [57] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The semantic web revisited. *IEEE Intelligent Systems*, 21(3):96–101, May 2006.
- [58] Yi Shang and Longzhuang Li. Precision evaluation of search engines. *World Wide Web*, 5:159–173, 2002.
- [59] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.
- [60] OpenLink Software. Openlink virtuoso jena provider. <http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtJenaProvider>.

- [61] OpenLink Software. Openlink virtuoso universal server. <http://virtuoso.openlinksw.com/>.
- [62] W3C. LargeTriplestores - w3c wiki. <http://www.w3.org/wiki/LargeTripleStores>.
- [63] W. John Wilbur and Karl Sirokin. The automatic identification of stop words. *J. Inf. Sci.*, 18(1):45–55, January 1992.

Glossary

A*

An approximative breadth-first search algorithm that replies on heuristics. [i](#), [62](#), [63](#), [94](#)

AllegroGraph

A high-performance, persistent database, produced by Franz Inc. [17](#)

bag-of-words

The decomposition of one or more documents into a set of distinct terms. [63](#), [67](#), [72](#), [81](#)

Berkeley Software Distribution

A Unix operating system derivative developed and distributed by the Computer Systems Research Group (CSRG) of the University of California, Berkeley, from 1977 to 1995.. [24](#), [25](#), [97](#)

Cascading Style Sheets

A formating language used extensively on web-sites. [43](#)

data mining

The practice of searching through large amounts of computerized data to find useful patterns or trends. [5](#)

DBpedia

A crowd-sourced community extracted graph based on Wikipedia[1]. [22](#), [46](#), [55](#), [56](#), [81](#), [84](#), [85](#), [94](#), [110](#)

directed graph

A graph in which each connection has a direction, leading from A to B, where the direction of a connection is commonly represented as an arrow. A directed

connection indicates either ownership (A owns B, A has B) or A has a one-way connection to B. [5](#)

Fuseki

An Apache Jena triple store with REST-style SPARQL access[39]. [22](#), [23](#), [101](#)

gFacet

Graph-based Faceted Exploration of RDF Data[34]. [4](#)

GitHub

A web-based hosting service for software development projects.. [17](#)

gold standard

A hand-annotated corpus of documents used for measuring performance of search algorithms. [75](#), [76](#), [88](#)

Google Guava

A Java library developed by Google that deals with: collections, caching, primitives support, concurrency libraries, common annotations, string processing and I/O[16]. [64](#), [71](#)

Gruff

A grapher-based triple-store browser produced by Franz Inc, for AllegroGraph,. [17](#)

information retrieval

Information retrieval is defined as finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers). [81](#)

Interesting Graph

An algorithm created by the research group to find and score interesting nodes in order to generate an interesting graph.. [i](#), [28](#), [54](#), [62–64](#), [67](#), [75](#), [81](#), [84](#), [88–90](#), [93](#), [94](#), [103](#)

Interesting Path

An algorithm created by the research group to find and score paths with a given start-node and end-node, containing interesting intermediate nodes.. [i](#), [29](#), [54](#), [67](#), [70](#), [75](#), [76](#), [88–90](#), [93](#), [94](#), [103](#)

Java

An object-oriented programming language managed by Oracle Inc.. [viii](#), [17–22](#), [24](#), [56](#), [80](#), [91](#), [97–99](#), [101](#)

Jena

A Java framework for working with RDF and RDFS[37]. [vii](#), [viii](#), [19–23](#), [25](#), [27–32](#), [35](#), [37](#), [44](#), [54–56](#), [87](#), [88](#), [93](#), [101](#), [102](#)

JGraphX

A Java visualisation API[21]. [vii](#), [viii](#), [24](#), [25](#), [27–29](#), [39](#), [41](#), [43](#), [44](#), [87](#), [93](#)

lazy evaluation

StringTemplate variables are lazily evaluated in the sense that referencing attribute "a" does not actually invoke the data lookup mechanism until the template is asked to render itself to text [30]. [98](#)

mxCell

An edge or vertex within an JGraphX graph. [39](#)

mxGraph

The main component of a JGraphX graph which contains all elements of the graph. [39](#), [44](#)

N-ary Tree

A tree where nodes can have an arbitrary degree. [64](#)

N-Triples

N-Triples are a subset of Turtle and thereby has a simpler format than Turtle. This makes N-Triples documents easier to parse and generate. [4]. [5](#)

Notation 3

This is a language which is a compact and readable alternative to RDF's XML syntax, but also is extended to allow greater expressiveness [5]. [5](#), [22](#)

Open Link Virtuoso

An open source triple store for RDF and RDFS[61]. [28](#), [110](#)

OWL 2 Web Ontology Language

The OWL Web Ontology Language is a language for defining and instantiating Web ontologies [6]. [5](#), [22](#)

OWL Web Ontology Language

An extension of OWL [7]. [5](#), [17](#), [21](#), [22](#), [81](#)

precision

The fraction of retrieved documents that are relevant. [i](#), [75–77](#), [82](#), [84](#), [88](#), [94](#)

Prefuse

A Java-based open-source visualisation framework. [25](#)

recall

The fraction of relevant documents that are retrieved. [75](#), [76](#)

REST

Representational State Transfer is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed hypermedia system.. [101](#)

SDB

A SQL Database back-end for Jena[37]. [22](#)

Semantic Web

A network of loosely connected entities, sharing semantically rich information, based on a common reference format. [vii](#), [viii](#), [3](#), [5](#), [16](#), [20](#), [93](#), [95](#)

Stanford Core NLP

A set of tools developed by the Stanford Natural Language Processing group for working with natural language[29]. [72](#)

StringTemplate

A Java library for creating and filling in templates for generating formatted text strings. See App. A. [55](#), [69](#), [70](#), [97–99](#)

Swing

A native Java GUI framework, which provides basic GUI objects, events and other logistical tools for the design and implementation of GUIs. [19](#), [24](#), [25](#), [27](#), [39](#), [47](#), [88](#), [128](#)

SwingWorker

A standard [Swing](#) class designed ease-of-multithreading for GUIs. [109](#)

TDB

Jena TDB is an file-based triple store[40]. [22](#), [58](#), [62](#), [64](#), [71](#), [87](#), [92](#), [101–103](#)

Triplestore

A database aimed towards storage of triples. The Triplestore exposes it's triples via query languages, such as SPARQL, and API calls. [11](#), [16](#), [17](#), [19](#), [21–23](#), [25](#), [28](#), [35–38](#), [54](#), [67](#), [81](#), [84](#), [85](#), [88](#), [93–95](#), [109](#)

Turtle

A Turtle(The Terse RDF Triple Language) document allows writing down an RDF graph in a compact textual form. It consists of a sequence of directives, triple-generating statements or blank lines. [2]. [5](#), [12](#), [22](#)

vector space model

A model for representing text as weighted terms. [81](#), [89](#)

Welkin

Welkin in the term describing the celestial sphere. [18](#)

Acronyms

API	Application Programming Interface. 17 , 18 , 21–23 , 25 , 29 , 101 , 102
BeFS	Best-First-Search. 62 , 64
BrFS	Breadth-First-Search. 33 , 34 , 37 , 62 , 64 , 85 , 86 , 91 , 94
ePOOLICE	early Pursuit against Organized crime using envirOnmental scanning, the Law and Intelli- genCE systems. 4
GUI	Graphical User Interface. 4 , 17 , 27 , 30 , 31 , 44 , 87
JUNG	Java Universal Network/Graph Framework. 17 , 24
JVM	Java Virtual Machine. 18 , 61
MRSW	Multiple-Reader/Single-Writer. 102
PBBBFS	Parallel Bucket-Based Breadth First Search. 56–60 , 64 , 75 , 84–86 , 89 , 91–95 , 112
POS	Part-Of-Speech. 89
RDF	Resource Description Framework. i , 5–8 , 12 , 13 , 17 , 19–23 , 25 , 27–30 , 37 , 62 , 67 , 69 , 87–90 , 93–95 , 101
RDF-Gravity	RDF GRAph VIualization Tool. 17 , 18
RDF/XML	Resource Description Framework Extensible Markup Language. 5 , 22

RDQL	RDF Data Query Language. 18
Rio	RDF/IO. 23
Sail	Storage And Inference Layer. 23
SCNLP	Stanford Core NLP. 89 , 90 , 105
SOAP	Simple Object Access Protocol. 35
SPARQL	SPARQL Protocol and RDF Query Language. vii , 3 , 4 , 8–17 , 19 , 21 , 22 , 25 , 28 , 31 , 33 , 35–37 , 44 , 51 , 54–56 , 58 , 61 , 62 , 64 , 67 , 69–71 , 88 , 91 , 93 , 98 , 99 , 101 , 102 , 109 , 110
SQL	Structured Query Language. 3 , 8 , 11 , 15
URI	Uniform Resource Identifier. 5 , 6 , 12 , 14 , 29 , 44 , 46 , 47 , 51 , 54–56 , 58 , 61 , 64 , 69 , 71 , 72 , 85 , 86 , 88 , 90 , 91 , 94 , 98 , 109
URL	Uniform Resource Locator. 31 , 35
W3C	World Wide Web Consortium. 5 , 8
WWW	World Wide Web. 3 , 5 , 95
XML	Extensible Markup Language. 40 , 47 , 103